Lexical elements

See also

These topics provide a formal definition of the Borland C++ lexical elements. They describe the different categories of word-like units (tokens) recognized by a language.

See the topics listed under **See Also** to learn about lexical elements.

The tokens in Borland C++ are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

A Borland C++ program starts as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Borland C++ editor). The basic program unit in Borland C++ is the file. This usually corresponds to a named file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see Preprocessor directives for details). For example, the directive **#include** *<inc_file>* adds (or *includes*) the contents of the file *inc_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and whitespace.

Whitespace

See also

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i;
   float f;
```

are lexically equivalent and parse identically to give the six tokens:

- **int**
- *i*
- **;**
- **float**
- **f**
- **;**

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process (they remain as part of the string). For example,

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International"

**Line splicing with **

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \
International"
```

is parsed as "Borland International" (see String constants for more information).

Comments

See also

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. Both are supported by Borland C++, with an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

You should also follow the guidelines on the use of whitespace and delimiters in comments discussed later in this topic to avoid other portability problems.

C comments

A C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurrence of the pair ***/** following the initial /*. The entire sequence, including the four comment-delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

Borland C++ does not support the nonportable *token pasting* strategy using /**/. Token pasting in Borland C++ is performed with the ANSI-specified pair ##, as follows:

```
#define VAR(i,j)   (i/**/j)     /* won't work */
#define VAR(i,j)   (i##j)       /* OK in Borland C++ */
#define VAR(i,j)   (i ## j)     /* Also OK */
```

In Borland C++,

```
int /* declaration */ i /* counter */;
```

parses as these three tokens:

```
int   i;
```

See Token Pasting with ## for a description of *token pasting.*

C++ comments

C++ allows a single-line comment using two adjacent slashes (//). The comment can start in any position, and extends until the next new line:

```
class X {  // this is a comment
... };
```

You can also use // to create comments in C code. This is specific to Borland C++.

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/*  int /* declaration */ i /* counter */; */
```

fails, because the scope of the first ***/*** ends at the first */. This gives

```
i ; */
```

which would generate a syntax error.

By default, Borland C++ won't allow nested comments, but you can override this with compiler options. See Options|Project|Compiler|Source|Nested Comments for information on enabling nested comments.

Delimiters and whitespace

In rare cases, some whitespace before /* and //, and after */, although not syntactically mandatory, can avoid portability problems. For example, this C++ code:

```
int i = j//* divide by k*/k;
+m;
```

parses as int i = j +m; not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

Tokens

See also

Tokens are word-like units recognized by a language. Borland C++ recognizes six classes of tokens.

Here is the formal definition of a token:

- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *operator*
- *punctuator* (also known as separators)

As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, *external* would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al.*

See Token Pasting with ## for a description of *token pasting.*

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names. See the:

- Alphabetical list of keywords.
- Table of C++ Keywords
- Table of Borland C++ Register Pseudovariables

You can use options to select ANSI keywords only, UNIX keywords, and so on; see Options|Project|Compiler|Source|Language Compliance for information on these options.

If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version.

Note: Note that the keywords **_ _try** and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword in the C++ exception-handling mechanism. **try** cannot be substituted by **_ _try.** The keyword **_ _try** can only be used to match the **_ _except** or **_ _finally** keywords. See the discussions on C++ exception handling and C-based structured exceptions for more information.

Identifiers

See also

Here is the formal definition of an identifier:

identifier:

nondigit

identifier nondigit

identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. (Identifiers can contain the letters *a* to *z* and *A* to *Z*, the underscore character "_", and the digits 0 to 9.) There are only two restrictions:

- The first character must be a letter or an underscore.
- By default, Borland C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu and command-line options, but not increased. See Options|Project|Compiler|Source|Identifier Length for information on these options.

Case sensitivity

Borland C++ identifiers are case sensitive, so that *Sum*, *sum* and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, Borland C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals *Sum* and *sum* are considered identical, resulting in a possible. "`Duplicate symbol`" warning during linking.

An exception to these rules is that identifiers of type **_ _pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are legal for *different* name spaces regardless of scope rules.

Constants

See also

Constants are tokens representing fixed numeric or character values.

Borland C++ supports four classes of constants: integer, floating point, character (including strings), and enumeration.

Internal representation of numerical types shows how these types are represented internally.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table.

Constants: Formal Definitions

constant:
  floating-constant
  integer-constant
  numeration-constant
  character-constant

floating-constant:
  fractional-constant <*exponent-part*> <*floating-suffix*>
  digit-sequence *exponent-part* <*floating-suffix*>

fractional-constant:
  <*digit-sequence*> **.** *digit-sequence*
  digit-sequence **.**

exponent-part:
  e <*sign*> *digit-sequence*
  E <*sign*> *digit-sequence*

sign: one of
  + -

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  f l F L

integer-constant:
  decimal-constant <*integer-suffix*>
  octal-constant <*integer-suffix*>
  hexadecimal-constant <*integer-suffix*>

decimal-constant:
  nonzero-digit
  decimal-constant *digit*

octal-constant:
  0

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

octal-digit: one of
  0 1 2 3 4 5 6 7

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix <*long-suffix*>
  long-suffix <*unsigned-suffix*>

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

enumeration-constant:
  identifier

character-constant
  c-char-sequence

c-char-sequence:
  c-char
  c-char-sequence *c-char*

c-char:
  Any character in the source character set

octal-constant *octal-digit*

hexadecimal-constant:

0 x hexadecimal-digit

0 X *hexadecimal-digit*

hexadecimal-constant *hexadecimal-digit*

except the single-quote ('), backslash (\), or newline character *escape-sequence.*

escape-sequence: one of the following

\"	\'	\?	\\
\a	\b	\f	\n
\o	\oo	\ooo	\r
\t	\v	\Xh...	\xh...

Integer constants

See also

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Borland C++ integer constants without L or U.. Note that the rules vary between decimal and nondecimal constants.

Decimal

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10;  /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0;   /*decimal 0 = octal 0 */
```

Octal

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces the constant to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul, lu, UL*, and so on. See the table of Borland constants.

The data type of a constant in the absence of any suffix (*U, u, L,* or *l*) is the first of the following types that can accommodate its value:

Decimal	**int, long int, unsigned long int**
Octal	**int, unsigned int, long int, unsigned long int**
Hexadecimal	**int, unsigned int, long int, unsigned long int**

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int, unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int, unsigned long int** that can accommodate its value.

If the constant has both u and l suffixes, (*ul, lu, Ul, lU, uL, Lu, LU or UL*), its data type will be **unsigned long int**.

Borland C++ integer constants without L or U summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Extended integer types

See also

You can specify the size for integer types. You must use the appropriate suffix when using extended integers.

Type	Suffix	Example	Storage
__int8	i8	__int8 c = 127i8;	8 bits
__int16	i16	__int16 s = 32767i16;	16 bits
__int32	i32	__int32 i = 123456789i32;	32 bits
__int64	i64	__int64 big = 12345654321i64;	64 bits
unsigned __int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64 bits

Borland C++ integer constants without L or U

See also

Decimal constants

0	to	32,767	**int**
32,768	to	2,147,483,647	**long**
2,147,483,648	to	4,294,967,295	**unsigned long**
		> 4294967295	truncated

Octal constants

00	to	077777	**int**
010000	to	0177777	**unsigned int**
02000000	to	017777777777	**long**
020000000000	to	037777777777	**unsigned long**
		> 037777777777	truncated

Hexadecimal constants

0x0000	to	0x7FFF	**int**
0x8000	to	0xFFFF	**unsigned int**
0x10000	to	0x7FFFFFFF	**long**
0x80000000	to	0xFFFFFFFF	**unsigned long**
		>0xFFFFFFFF	truncated

Floating-point constants

See also

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- *e* or *E* and a signed integer exponent (optional)
- Type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Here are some examples:

Constant	Value
23.45e6	23.45 106
.0	0
0.	0
1.	1.0 100 = 1.0
-1.23	-1.23
2e-5	2.0 10-5
3E+10	3.0 1010
.09E34	0.09 1034

In the absence of any suffixes, floating-point constants are of type **double.** However, you can coerce a floating constant to be of type **float** by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The table below shows the ranges available for **float**, **double**, and **long double**.

Borland C++ floating-point constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4 10^{-38} to 3.4 10^{38}
double	64	1.7 10^{-308} to 1.7 10^{308}
long double	80	3.4 10^{-4932} to 1.1 10^{4932}

Character constants

See also

A *character constant* is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-charactrer constants have data type **int**. The number of bits used to internally represent a character constant is **sizeof**(**int**). In a 16-bit program, the upper byte is zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

To learn more about character constants, see

- Three char types
- Escape sequences
- Wide-character and multi-character constants

Note: To compare sizes of character types, compile this as a C program and then as a C++ program.

```
#include <stdio.h>
#define CH 'x'       /* A CHARACTER CONSTANT */
void main(void) {
   char ch = 'x';    /* A char VARIABLE      */
   printf("\nSizeof int     = %d", sizeof(int) );
   printf("\nSizeof char    = %d", sizeof(char) );
   printf("\nSizeof ch      = %d", sizeof(ch) );
   printf("\nSizeof CH      = %d", sizeof(CH) );
   printf("\nSizeof wchar_t = %d", sizeof(wchar_t) );
}
```

Note: Sizes are in bytes.

Sizes of character types

Output when compiled as C program			Output when compiled as C++ program		
	16-bit	32-bit		16-bit	32-bit
Sizeof int =	2	4	Sizeof int=	2	4
Sizeof char =	1	1	Sizeof char =	1	1
Sizeof ch =	1	1	Sizeof ch =	1	1
Sizeof CH =	2	4	Sizeof CH =	1	1
Sizeof wchar_t =	2	2	Sizeof wchar_t =	2	2

The three char types

See also

One-character constants, such as 'A', '\t' and '007', are represented as **int** values. In this case, the low-order byte is *sign extended* into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to -1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned**, which forces the high bit to be zero regardless of the value of the low bit. See Options|Project|C++ Options|C++ Compatibility|Do not treat 'char' as distinct type for information on these options.

The three character types, **char**, **signed char**, and **unsigned char,** require an 8-bit (one byte) storage. In C and Borland C++ programs prior to version Borland C++ 4.0 , **char** is treated the same as **signed char**. The behavior of C programs is unaffected by the distinction between the three character types.

Note: To retain the old behavior, use the -K2 command-line option and Borland C++ 3.1 header files and libraries.

In a C++ program, a function can be overloaded with arguments of type **char**, **signed char**, or **unsigned char**. For example, the following function prototypes are valid and distinct:

```
void func(char ch);
void func(signed char ch);
void func(unsigned char ch);
```

If only one of the above prototypes exists, it will accept any of the three character types. For example, the following is acceptable:

```
void func(unsigned char ch);
void main(void) {
  signed char ch = 'x';
  func(ch);
  }
```

See Options|Project|Compiler|Code Generation for a description of code-generation options.

Escape sequences

See also

The backslash character (\) is used to introduce an *escape sequence*, which allows the visual representation of certain nongraphic characters. For example, the constant **\n** is used to the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for *Ctrl-C* or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for Borland C++). Larger numbers generate the compiler error Numeric constant too large. For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Originally, Turbo C allowed only three digits in a hexadecimal escape sequence. The ANSI C rules adopted in Borland C++ might cause problems with old code that assumes only the first three characters are converted. For example, using Turbo C 1.

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, Borland C++ compiles it as the hexadecimal number \x0072 and the literal string "2.1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The following table shows the available escape sequences.

Borland C++ escape sequences

Note: You must use \\ to represent an ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O=a string of up to three octal digits
\xH		any	H=a string of hex digits
\XH		any	H=a string of hex digits

Wide-character and multi-character constants

See also

Wide-character types can be used to represent a character that does not fit into the storage space allocated for a **char** type. A wide character is stored in a two-byte space. A character constant preceded immediately by an *L* is a wide-character constant of data type *wchar_t* (defined in stddef.h). For example:

```
wchar_t ch = L'AB';
```

When *wchar_t* is used in a C program it is a type defined in stddef.h header file. In a C++ program, **wchar_t** is a keyword that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In C++, **wchar_t** is the same size, signedness, and alignment requirement as an **int** type.

A string preceded immediately by an *L* is a wide-character string. The memory allocation for a string is two bytes per character. For example:

```
wchar_t str = L"ABCD";
```

Multi-character constants

Borland C++ also supports multi-character constants. When using the 32-bit compiler, multi-character constants can consist of as many as four characters. The 16-bit compiler is restricted to two-character constants. For example, 'An', '\n\t', and '\007\007' are acceptable in a 16-bit program. The constant, '\006\007\008\009' is valid only in a 32-bit program. When using the 16-bit compiler, these constants are represented as 16-bit int values with the first character in the low-order byte and the second character in the high-order byte. For 32-bit compilers, multi-character constants are always 32-bit **int** values. The constants are not portable to other C compilers.

String constants

See also

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of-**char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences. This code, for example:

```
"\t\t\"Name\"\\\tAddress\n\n"
```

prints like this:

```
"Name"\        Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \" provides interior double quotes.

If you compile with the -**A** option for ANSI compatibility, the escape character sequence "\\", is translated to "\" by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>
int main() {
   char    *p;
   _InitEasyWin();
   p = "This is an example of how Borland C++"
     " will\nconcatenate very long strings for you"
     " automatically, \nresulting in nicer"
      " looking programs.";
   printf(p);
   return(0);
}
```

The output of the program is

```
This is an example of how Borland C++ will
concatenate very long strings for you automatically,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

Enumeration constants

See also

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, *cubs,* and *dodgers* are enumeration constants of type *team* that can be assigned to any variables of type *team* or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

Constants and internal representation

See also

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For Borland C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of internal representations for the various data types.

The following tables list the sizes and resulting ranges of the data types for Borland C++. Internal representation of numerical types shows how these types are represented internally.

16-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	16	-32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4 ^ 10-38 to 3.4 ^ 1038	Scientific (7-digit precision)
double	64	1.7 ^ 10-308 to 1.7 ^ 10308	Scientific (15-digit precision)
long double	80	3.4 ^ 10-4932 to 1.1 ^ 104932	Financial (18-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

32-bit data types, sizes, and ranges

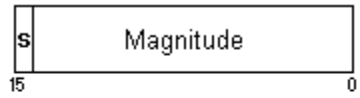
Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned int	32	0 to 4,294,967,295	Large numbers and loops
int	32	-2,147,483,648 to 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
enum	32	-2,147,483,648 to 2,147,483,647	Ordered sets of values
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4 ^ 10-38 to 1.7 ^ 1038	Scientific (7-digit) precision)
double	64	1.7 ^ 10-308 to 3.4 ^ 10308	Scientific (15-digit precision)
long double	80	3.4 ^ 10-4932 to 1.1 ^ 104932	Financial (18-digit precision)

Internal representation of numerical types

See also

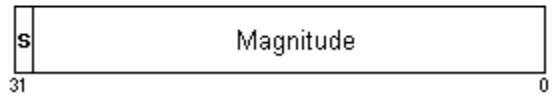
16-bit integers

int



(2's complement)

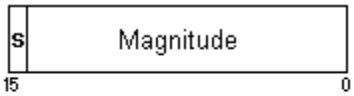
long int



(2's complement)

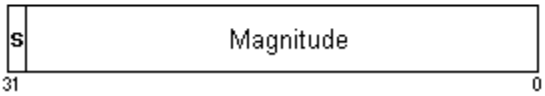
32-bit integers

short int



(2's complement)

int, long int

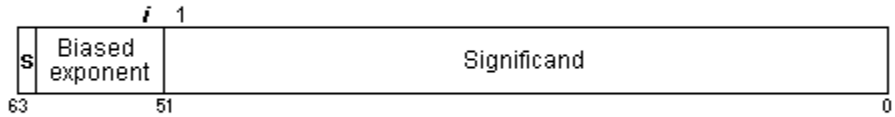


(2's complement)

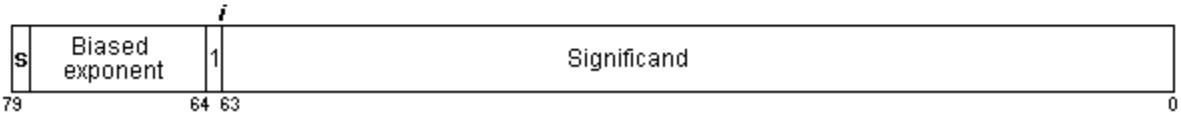
Floating-point types, always

float

double



long double



s	=	Sign bit (0 = positive, 1 = negative)	Exponent bias (normalized values):	
i	=	Position of implicit binary point	**float:**	127 (7FH)
1	=	Integer bit of significance:	**double:**	1,023 (3FFH)
		Stored in **long double** Implicit in **float**, **double**	**long double:**	16,383 (3FFFH)

Constant expressions

See also

A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is:

constant-expression:

Conditional-expression

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- Assignment
- Comma
- Decrement
- Function call
- Increment

Punctuators

See also

The punctuators (also known as separators) in Borland C++ are defined as follows:

punctuator: one of

[ ] ( ) { } , ; : ... * = #

Brackets

Open and close brackets **[ ]** indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];           /* 3 x 4 matrix */
ch = str[3];             /* 4th element */
  .
  .
  .
```

Parentheses

Open and close parentheses **( )** are used to group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);     /* override normal precedence */
if (d == z) ++x;     /* essential with conditional statement */
func();  /* function call, no args */
int (*fptr)();       /* function pointer declaration */
fptr = func;         /* no () means func pointer */
void func2(int n);   /* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered in Expressions.

Braces

Open and close braces **{ }** indicate the start and end of a compound statement:

```
if (d == z)
{
   ++x;
   func();
}
```

The closing brace serves as a terminator for the compound statement, so a ; (semicolon) is not required after the }, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
   {};                  /*illegal semicolon*/
else
```

Comma

The comma (**,**) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j);                              /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5));  /* also calls func with two args! *
  /
```

Semicolon

The semicolon (**;**) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, Borland C++ might ignore it.

```
a + b;     /* maybe evaluate a + b, but discard value */
++a;       /* side effect on a, but discard value of ++a */
;          /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement:*

```
for (i = 0; i < n; i++)
{
   ;
}
```

Colon

Use the colon (**:**) to indicate a labeled statement:

```
start:    x=0;
   f
goto start;
```

Labels are discussed in Labeled statements.

Ellipsis

The ellipsis (**...**) is three successive periods with no intervening whitespace. Ellipses are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that *func* will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.

In C++, you can omit the comma before the ellipsis.

Asterisk (pointer declaration)

The asterisk (*****) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr;  /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;          /* a pointer to an integer array */
double ***double_ptr;   /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer)

The equal sign (**=**) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int  x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }  /* Parameter i has default value of zero */
```

The equal sign is also used as the assignment operator in expressions:

```
int a, b, c;
a = b + c;
float *ptr = (float *) malloc(sizeof(float) * 100);
```

Pound sign (preprocessor directive)

The pound sign (**#**) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See Preprocessor directives for more on the preprocessor directives.

and **##** (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

Language structure

See also

These topics provide a formal definition of Borland C++ language structure. They describe the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

Declarations

See also

This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

See also

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely "points" to the object. The type is used

- To determine the correct memory allocation required initially.
- To interpret the bit patterns found in the object during subsequent accesses.
- In many type-checking situations, to ensure that illegal assignments are trapped.

Borland C++ supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in various memory models.

The Borland C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Borland C++ can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Objects and declarations

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this rule (known as *forward references*) are labels, calls to undeclared functions, and class, struct, or union tags

lvalues

An *lvalue* is an object locator: an expression that designates an object. An example of an lvalue expression is **P*, where *P* is any expression evaluating to a non-null pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the *l* stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if *a* and *b* are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as *a* = *1*; and *b* = *a* + *b* are legal.

rvalues

The expression *a* + *b* is not an lvalue: *a* + *b* = *a* is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

Storage classes and types

See also

Associating identifiers with objects requires each identifier to have at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Borland C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type. See sizeof for more on this operator.

Scope

See also

The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

- **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.
- **Function**. The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label_name:* followed by a statement. Label names must be unique within a function.
- **Function prototype**. Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- **File**. File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- **Class** (C++). A class is a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class.Classes and their objects have many special access and scoping rules; see Classes.
- **Condition** (C++). Declarations in conditions are supported. Variables can be declared within the expression of **if**, **while**, and **switch** statements. The scope of the variable is that of the statement. In the case of an **if** statement, the variable is also in scope for the **else** block.

Name spaces

Name space is the scope within which an identifier must be unique. C uses four distinct classes of identifiers:

- **goto** label names. These must be unique within the function in which they are declared.
- Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all
- Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
- Variables, **typedef**s, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Note: Structures, classes, and enumerations are in the same name space in C++.

Visibility

See also

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Note: Visibility cannot exceed scope, but scope can exceed visibility.

```
     .
     .
     .
{
   int i; char ch;  // auto by default
   i = 3;           // int i and char ch in scope and visible
     .
     .
     .

   {
      double i;
      i = 3.0e3;    // double i in scope and visible
                    // int i=3 in scope but hidden
      ch = 'A';     // char ch in scope and visible
   }
                    // double i out of scope
   i += 1;          // int i visible and = 4
     .
     .
     .
// char ch still in scope & visible = 'A'
}
     .
     .
     .
// int i and char ch out of scope
```

Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be accessed under certain conditions

Duration

See also

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedef**s and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

Static

Memory is allocated to objects with *static* duration as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Don't confuse static duration with file or global scope. An object can have static duration and local scope

Local

Local duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects must always have local or function scope. The storage class specifier **auto** can be used when declaring local duration variables, but is usually redundant, because **auto** is the default for variables declared within a block. An object with local duration also has local scope, because it does not *exist* outside of its enclosing block. The converse is not true: a local scope object can have static duration.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. Borland C++ can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Note: The Borland C++ compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Dynamic

Dynamic duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the heap, using either standard library functions such as *malloc*, or by using the C++ operator **new**. The corresponding deallocations are made using *free* or **delete**.

Translation units

See also

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

translation-unit:

external-declaration

translation-unit external-declaration

external-declaration

function-definition

declaration

word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the section Linkage.) Any declaration that also reserves storage for an object or function is called a definition (or defining declaration). For more details, see External declarations and definitions.

Linkage

See also

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function within one file only. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules

- Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.
- For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.
- If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
- If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
- If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

Identifiers with no linkage attribute:

- Any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
- Function parameters
- Block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .OBJ files compiled with a C compiler

To tell the C++ compiler not to mangle the name of a function, declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler that references to the function *Cfunc* should not be mangled.

You can also apply the `extern "C"` declaration to a block of names:

```
extern "C" {
   void Cfunc1( int );
   void Cfunc2( int );
   void Cfunc3( int );
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions *Cfunc1*, *Cfunc2*, and *Cfunc3* should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {
   #include "locallib.h"
};
```

Introduction to declaration syntax

See also

All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative definitions

See also

The ANSI C standard supports the concept of the *tentative definition*. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;
int x;          /*legal, one copy of x is reserved */
int y;
int y = 4;      /* legal, y is initialized to 4 */
int z = 5;
int z = 6;      /* not legal, both are initialized definitions */
```

Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible declarations

See also

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use **typedefs** to improve legibility.

In Borland C++ declaration syntax, note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail in Variable Modifiers, Pointer Modifiers, and Function Modifiers.

Borland C++declaration syntax

declaration:
<decl-specifiers> <declarator-list>;
asm-declaration
function-declaration
linkage-specification

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
friend (C++ specific)
typedef

decl-specifiers:
<decl-specifiers> decl-specifier

storage-class-specifier:
auto
register
static
extern

function-specifier: (C++ specific)
inline
virtual

simple-type-name:
class-name
***typedef**-name*
char
short
int

elaborated-type-specifier:
class-key identifier
class-key class-name
enum *enum-name*

class-key: (C++ specific)
class
struct
union

enum-specifier:
enum *<identifier>* { *<enum-list>* }

enum-list:
enumerator
enumerator-list , enumerator

enumerator:
identifier
identifier = constant-expression

constant-expression:
conditional-expression

linkage-specification: (C++ specific)
extern *string* { *<declaration-list>* }
extern *string declaration*

type-specifier:
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
const

long
signed
unsigned
float
double
void

declarator-list:
init-declarator
declarator-list **,** *init-declarator*

init-declarator:
declarator <*initializer*>

declarator:
dname
modifier-list
pointer-operator declarator
declarator (*parameter-declaration-list*)
<*cv-qualifier-list* >
(The <cv-qualifier-list > is for C++ only.)
declarator [<*constant-expression*>]
(*declarator*)

modifier-list:
modifier
modifier-list *modifier*

modifier:
_ _cdecl
_ _pascal
_ _interrupt
_ _near
_ _far
_ _huge

pointer-operator:
***** <*cv-qualifier-list*>
& <*cv-qualifier-list*> (C++ specific)
class-name **::** ***** <*cv-qualifier-list*>
(C++ specific)

cv-qualifier-list:
cv-qualifier <*cv-qualifier-list*>

cv-qualifier
const
volatile

dname:
name
class-name (C++ specific)
~ *class-name* (C++ specific)
type-defined-name

volatile

declaration-list:
declaration
declaration-list ; declaration

type-name:
type-specifier <*abstract-declarator*>

abstract-declarator:
pointer-operator <*abstract-declarator*>
<*abstract-declarator*> (*argument-declaration-list*)
<*cv-qualifier-list*>
<*abstract-declarator*> [<*constant-expression*>]
(*abstract-declarator*)

argument-declaration-list:
<*arg-declaration-list*>
arg-declaration-list **, ...**
<*arg-declaration-list*> **...** (C++ specific)

arg-declaration-list:
argument-declaration
arg-declaration-list **,** *argument-declaration*

argument-declaration:
decl-specifiers declarator
decl-specifiers declarator = *expression*
(C++ specific)
decl-specifiers <*abstract-declarator*>
decl-specifiers <*abstract-declarator*> **=** *expression*
(C++ specific)

function-definition:

function-body:
compound-statement

initializer:
= *expression*
= { *initializer-list* }
(*expression-list*) (C++ specific)

initializer-list:
expression
initializer-list **,** *expression*
{ *initializer-list* <**,**> }

External declarations and definitions

See also

The storage class specifiers auto and register cannot appear in an external declaration. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of sizeof), then exactly one external definition of that identifier must be somewhere in the entire program.

Borland C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];              // no size
struct mystruct;      // tag only, no member declarators
  .
  .
  .
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
   int i, j;
};                    // add member declarators
```

Borland C++ class declaration syntax (C++ only) covers class declaration syntax. In the section on classes (beginning with Classes), you can find examples of how to declare a class. Referencing covers C++ reference types (closely related to pointer types) in detail. Finally, see Using Templates for a discussion of **template**-type classes.

Borland C++ class declaration syntax (C++ only)

class-specifier:
  class-head { *<member-list>* }
class-head:
  class-key <identifier> <base-specifier>
  class-key class-name <base-specifier>
member-list:
  member-declaration <member-list>
  access-specifier : <member-list>
member-declaration:
  <decl-specifiers> <member-declarator-list> ;
  function-definition <;>
  qualified-name ;
member-declarator-list:
  member-declarator
  member-declarator-list, member-declarator
member-declarator:
  declarator <pure-specifier>
  <identifier> : constant-expression
pure-specifier:
  = 0
member-initializer-list:
  member-initializer
  member-initializer , member-initializer-list
member-initializer:

base-specifier:
  : *base-list*
base-list:
  base-specifier
  base-list , base-specifier
base-specifier:
  class-name
  virtual *<access-specifier> class-name*
  access-specifier **<virtual>** *class-name*
access-specifier:
  private
  protected
  public
conversion-function-name:
  operator *conversion-type-name*
conversion-type-name:
  type-specifiers <pointer-operator>
constructor-initializer:
  : *member-initializer-list*

operator-name: one of
  new delete sizeof typeid
+ - * / % ^
& | ~ ! = <>

class name (<*argument-list*>)	**+=**	**-=**	**=***	**/=**	**%=**	**^=**
identifier (<*argument-list*>)	**&=**	**|=**	**<<**	**>>**	**>>=**	**<<=**
operator-function-name:	**==**	**!=**	**<=**	**>=**	**&&**	**||**
operator *operator-name*	**++**	**__**	**,**	**->***	**->**	**()**
	[ ]	**.***				

Type categories

See also

The four basic type categories (and their subcategories) are as follows:

- Aggregate
- Array
- **struct**
- **union**
- **class** (C++ only)
- Function
- Scalar
- Arithmetic
- Enumeration
- Pointer
- Reference (C++ only)
- void)

Types can also be viewed in another way: they can be *fundamental* or *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.

A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Declaring types

Declaration	Description
type *t;*	An object of type ***type***
type *array*[10];	Ten ***type***s: *array*[0] - *array*[9]
type **ptr;*	*ptr* is a pointer to ***type***
type *&ref* = *t;*	*ref* is a reference to ***type*** (C++)
type *func*(**void**);	*func* returns value of type ***type***
void *func1*(***type*** *t*);	*func1* takes a type ***type*** parameter
struct st {***type*** *t1*; ***type*** *t2*};	structure **st** holds two ***type***s

Note: type& var, type &var, and type & var are all equivalent.

The fundamental types

See also

The fundamental type specifiers are built from the following keywords:

char	**__int8**	**long**
double	**__int16**	**signed**
float	**__int32**	**short**
int	**__int64**	**unsigned**

From these keywords you can build the integral and floating-point types, which are together known as the *arithmetic* types. The modifiers **long**, **short**, **signed**, and **unsigned** can be applied to the integral types. The header file limits.h contains definitions of the value ranges for all the fundamental types.

Integral types

char, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. Integral types shows the integral type specifiers, with synonyms listed on the same line.

Integral types

char, signed char	Synonyms if default **char** set to **signed.**
unsigned char	
char, unsigned char	Synonyms if default **char** set to **unsigned.**
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Note: These synonyms are not valid in C++. See The three char types.

Only **signed** or **unsigned** can be used with **char**, **short**, **int**, or **long**. The keywords **signed** and **unsigned**, when used on their own, mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Borland C++ lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares *ch* as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

Only **long** or **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to indicate that **short**, **int**, and **long** form a nondecreasing sequence with **"short <= int <= long."** All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In a Borland C++ 16-bit program, the types **int** and **short** are equivalent, both being 16 bits. In a Borland C++ 32-bit program, the types **int** and **long** are equivalent, both being 32 bits. The signed varieties are all stored in two's complement format using the most significant bit (MSB) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown in 16-bit data types, sizes, and ranges and 32-bit data types, sizes, and ranges). In the unsigned versions, all bits are used to give a range of 0 - (2*n* - 1), where *n* is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. Borland C++ uses the IEEE floating-point

formats.See the topic on ANSI implementation-specific.

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double** *test_case*, for example.

16-bit data types, sizes, and ranges and 32-bit data types, sizes, and ranges indicates the storage allocations for the floating-point types

Standard arithmetic conversions

When you use an arithmetic expression, such as *a* + *b*, where *a* and *b* are different arithmetic types, Borland C++ performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps Borland C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Methods used in standard arithmetic conversions. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers), or they are of type **double**, **float**, or **long double**.
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Methods used in standard arithmetic conversions

Type	Converts to	Method
char	**int**	Zero or sign-extended (depends on default char type)
unsigned char	**int**	Zero-filled high byte (always)
signed char	**int**	Sign-extended (always)
short	**int**	Same value; sign extended
unsigned short	**unsigned int**	Same value; zero filled
enum	**int**	Same value

Special char, int, and enum conversions

Note: The conversions discussed in this section are specific to Borland C++.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

Initialization

See also

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

- To zero if it is an arithmetic type
- To null if it is a pointer type

Note: If the object has automatic storage duration, its value is indeterminate.

Syntax for initializers

initializer

= *expression*

= {*initializer-list*} <,>}

(*expression list*)

initializer-list

expression

initializer-list, expression

{*initializer-list*} <,>}

Rules governing initializers

- The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
- The item to be initialized must be an object (for example, an array) of unknown size.
- For C (not required for C++), all expressions must be constants if they appear in one of these places:
- In an initializer for an object that has static duration.
- In an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed).
- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list (as described in Arrays, structures, and unions).
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, which counts how many times each day of the week appears in a month (assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

The following rules initialize character arrays and wide character arrays:

- You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for *name*[0]), 'n' (for *name*[1]), and

so on (and including a null terminator).

- You can initialize a wide character array (one that is compatible with *wchar_t*) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
   int i;
   char str[21];
   double d;
   } s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces.

Declarations and declarators

See also

A *declaration* is a list of names. The names are sometimes referred to as *declarators* or *identifiers*. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Simple declarations of variable identifiers have the following pattern:

data-type var1 <=init1>, var2 <=init2>, ...;

where *var1*, *var2*,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.

In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions

The format of the declarator indicates how the declared *name* is to be interpreted when used in an expression. If ***type*** is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

storage-class-specifier ***type*** *D1, D2;*

indicates that each occurrence of D1 or D2 in an expression will be treated as an object of type ***type*** and storage class *storage class specifier*. The type of the *name* embedded in the declarator will be some phrase containing ***type***, such as "***type***," "pointer to ***type***," "array of ***type***," "function returning ***type***," or "pointer to function returning ***type***," and so on.

For example, in Declaration syntax examples each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Declaration syntax examples

Declarator syntax	Implied *type of name*	Example
`type name;`	*type*	`int count;`
`type name[];`	(open) array of ***type***	`int count[];`
`type name[3];`	Fixed array of three elements, all of **type** (*name*[0], *name*[1], and *name*[2]	`int count[3];`
`type *name;`	Pointer to **type**	`int *count;`
`type *name[];`	(open) array of pointers to **type**	`int *count[];`
`type *(name[]);`	Same as above	`int *(count[]);`
`type (*name)[];`	Pointer to an (open) array of **type**	`int (*count) [];`
`type &name;`	Reference to **type** (C++ only)	`int &count;`
`type name();`	Function returning ***type***	`int count();`
`type *name();`	Function returning pointer to ***type***	`int *count();`
`type *(name());`	Same as above	`int *(count());`
`type (*name)();`	Pointer to function returning ***type***	`int (*count) ();`

Note the need for parentheses in (**name*)[ ] *and* (**name*)( ); this is because the precedence of both the

array declarator [ ] and the function declarator ( ) is higher than the pointer declarator *. The parentheses in *(*name*[ ]) are optional.

Note: See Borland C++declaration syntax for the declarator syntax. The definition covers both identifier and function declarators.

Variable modifiers

See also

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier. The modifiers available with Borland C++ are summarized in Borland C++ modifiers.

Mixed-language calling conventions

Borland C++ allows your programs to easily call routines written in other languages, and vice versa. When you mix languages , you have to deal with two important issues: identifiers and parameter passing.

By default, Borland C++ saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you can select the **-u** command-line option, or uncheck the compiler option setting in the IDE.

Note: The section Linkage tells how to use **extern**, which allows C names to be referenced from a C++ program.

Calling conventions summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the *caller*) or the called function (the *callee*) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.

Calling conventions

Modifier	Push parameters	Pop parameters	Name change
_ _cdecl[1]	Right first	Caller	'_' prepended
_ _fastcall	Left first	Callee	'@' prepended
_ _pascal	Left first	Callee	Uppercase
_ _stdcall	Right first	Callee	No change

1. This is the default.

Note: **__fastcall** and **_ _stdcall** are subject to name mangling. See the description of the -VC option .

Multithread variables

Keywords

The keyword **_ _thread** is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a **_ _thread** variable for each threaded process.

The syntax is *Type* **__thread** *variable__name*. For example

```
int __thread x;
```

declares an integer type variable that will be global but private to each thread in the program in which the statement occurs.

The **_ _thread** modifier can be used with global (file-scope) and static variables. The modifier cannot be used with pointers or functions. (However, you can have pointers to **_ _thread** objects.) A program element that requires run-time initialization or run-time finalization cannot be declared to be a **_ _thread** type. The following declarations require run-time initialization and are therefore illegal.

```
int f( );
int __thread x = f( );   // illegal
```

Instantiation of a class with a user-defined constructor or destructor requires run-time initialization and is therefore illegal.

```
class X  {
   X( );
   ~X( );
};
X __thread myclass;   // illegal
```

Pointer modifiers

See also

Borland C++ has modifiers that affect the pointer declarator (*); that is, they modify pointers to data. These are **_ _near**, **_ _far**, **_ _huge**, **_ _cs**, **_ _ds**, **_ _es**, **_ _seg**, and **_ _ss**.

You can compile a program using one of several memory models. The model you use determines (among other things) the internal format of pointers. For example, if you use a small data model(small or medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact or large), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes when you're using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the pointer modifiers.

See __near, __far, and __huge for an in-depth explanation of these types of pointers, and a description of normalized pointers. Also see the additional discussions of __cs, __ds, __es, __seg, and __ss.

Function modifiers

See also

This section presents descriptions of the Borland C++ function modifiers

In addition to their use as pointer modifiers, the **_ _near**, **_ _far**, and **_ _huge** modifiers can also be used as function type modifiers; that is, they can modify functions and function pointers as well as data pointers. In addition, you can use the **_ _loadds**, **_ _export**, **_ _import**, and **_ _saveregs** modifiers to modify functions.

Note: Tiny and huge memory models are not supported in Windows programs.

Also see Class memory model specifications.

In a 16-bit program, the **_ _import** can be used only as a modifier for class declarations. In 32-bit programs the keyword can be applied to class, function, and variable declarations

The **_ _near**, **_ _far**, and **_ _huge** function modifiers can be combined with **_ _cdecl** or **_ _pascal**, but not with **_ _interrupt**.

Functions of type **_ _huge** are useful when interfacing with code in assembly language that doesn't use the same memory allocation as Borland C++.

A function that is not an **_ _interrupt** type can be declared to be **_ _near**, **_ _far**, or **_ _huge** in order to override the default settings for the current memory model

A **_ _near** function uses **_ _near** calls; a **_ _far** or **_ _huge** function uses **_ _far** call instructions.

In the small and compact memory models, an unqualified function defaults to type **_ _near**. In the medium and large models, an unqualified function defaults to type **_ _far**.

A **_ _huge** function is the same as a **_ _far** function, except that the DS register is set to the data segment address of the source module when a **_ _huge** function is entered, but left unset for a **_ _far** function

The **_ _export** modifier makes the function exportable from Windows. The **_ _import** modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL; see Entry/Exit Code for details

The **_ _loadds** modifier indicates that a function should set the **DS_register**, just as a **_ _huge** function does, but does not imply **_ _near** or **_ _far** calls. Thus, **_ _loadds _ _far** is equivalent to **_ _huge**.

The **_ _saveregs** modifier causes the function to preserve all **register** values and restore them before returning (except for explicit return_values passed in registers such as AX or DX).

The **_ _loadds** and **_ _saveregs** modifiers are useful for writing low-level interface routines, such as mouse support routines.

Functions declared with the **_ _fastcall** modifier have different names than their non-**_ _fastcall** counterparts. The compiler prefixes the **_ _fastcall** function name with an @. This prefix applies to both unmangled C function names and to mangled C++ function names.

Borland C++ modifiers

Modifier	Use with	Description
const1	Variables	Prevents changes to object.
volatile1	Variables	Prevents register allocation and some optimization. Warns compiler that object might be subject to outside change during evaluation.
_ _cdecl2	Functions	Forces C argument-passing convention. Affects Linker and link-time names.
_ _cdecl2	Variables	Forces global identifier case-sensitivity and leading underscores.
_ _interrupt	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt

		handlers.
_ _pascal	Functions	Forces Pascal argument-passing convention. Affects Linker and link-time names.
_ _pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
_ _near,	Pointer types	Overrides the default pointer type specified by the current memory model.
_ _far,		
_ _huge		
_ _cs,	Pointer types	Segment pointers.
_ _ds,		
_ _es,		
_ _seg,		
_ _ss		
_ _near,	Functions	Overrides the default function type specified by the current memory model.
_ _far,		
_ _huge		
_ _near,	Variables	Directs the placement of the object in memory.
_ _far		
_ _export	Functions/classes	Tells the compiler which functions or classes to export.
_ _import	Functions/classes	Tells the compiler which functions or classes to import. (In 16-bit programs, this keyword can be used only for class declarations.)
_ _loadds	Functions	Sets DS to point to the current data segment.
_ _saveregs	Functions	Preserves all register values (except for return values) during execution of the function.
_ _fastcall	Functions	Forces register parameter passing convention. Affects the linker and link-time names.
_ _stdcall	Function	Forces the standard WIN32 argument-passing convention.

1 C++ extends **const** and **volatile** to include classes and member functions.

2 This is the default.

Pointers

See also

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain Borland C++ operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Note: See Referencing for a discussion of referencing and dereferencing.

Pointers to objects

See also

A pointer of type "pointer to object of ***type***" holds the address of (that is, points to) an object of ***type***. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

The size of pointers to objects is dependent on the memory model and the size and disposition of your data segments, possibly influenced by the optional pointer modifiers (discussed starting with Pointer modifiers)

Pointers to functions

See also

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called "pointer to function returning ***type***," where type is the function's return ***type***. For example,

```
void (*func)();
```

In C++, this is a pointer to a function taking no arguments, and returning **void**. In C, it's a pointer to a function taking an unspecified number of arguments and returning **void**. In this example,

```
void (*func)(int);
```

func* is a pointer to a function taking an **int argument and returning **void**.

For C++, such a pointer can be used to access static member functions. Pointers to class members must use pointer-to-member operators. See static_cast for details.

Pointer declarations

See also

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. Borland C++ lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to void. And in C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer. See void for details.

Warning! You need to initialize pointers before using them.

If ***type*** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr;    /* Uninitialized pointer */
```

declares *ptr* to be of type "pointer to ***type***." All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

The pointer type "pointer to **void**" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any "pointer to ***type***" value, including null, without complaint. Assignments without proper casting between a "pointer to ***type1***" and a "pointer to ***type2***," where ***type1*** and ***type2*** are different types, can invoke a compiler warning or error. If ***type1*** is a function and ***type2*** isn't (or vice versa), pointer assignments are illegal. If ***type1*** is a pointer to **void**, no cast is needed. Under C, if ***type2*** is a pointer to **void**, no cast is needed.

Assignment restrictions also apply to pointers of different sizes (**_ _near**, **_ _far**, and **_ _huge**). You can assign a smaller pointer to a larger one without error, but you can't assign a larger pointer to a smaller one unless you are using an explicit cast. For example,

```
char _ _near *ncp;
char _ _far  *fcp;
char _ _huge *hcp;
fcp = ncp;                    // legal
hcp = fcp;                    // legal
fcp = hcp;                    // not legal
ncp = fcp;                    // not legal
ncp = (char _ _near*)fcp;     // now legal
```

Pointer constants

See also

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```
int i;                          // i is an int
int * pi;                       // pi is a pointer to int (uninitialized)
int * const cp = &i;            // cp is a constant pointer to int
const int ci = 7;               // ci is a constant int
const int * pci;                // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
                                // constant int
```

The following assignments are legal:

```
i = ci;                       // Assign const-int to int
*cp = ci;                     // Assign const-int to
                              // object-pointed-at-by-a-const-pointer
++pci;                        // Increment a pointer-to-const
pci = cpc;                    // Assign a const-pointer-to-a-const to a
                              // pointer-to-const
```

The following assignments are illegal:

```
ci = 0;                       // NO--cannot assign to a const-int
ci--;                         // NO--cannot change a const-int
*pci = 3;                     // NO--cannot assign to an object
                              // pointed at by pointer-to-const
cp = &ci;                     // NO--cannot assign to a const-pointer,
                              // even if value would be unchanged
cpc++;                        // NO--cannot change const-pointer
pi = pci;                     // NO--if this assignment were allowed,
                              // you would be able to assign to *pci
                              // (a const value) by assigning to *pi.
```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

Pointer arithmetic

See also

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to ***type***" automatically take into account the size of ***type***; that is, the number of bytes needed to store a ***type*** object.

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to ***type***, adding an integral value to the pointer advances the pointer by that number of objects of ***type***. If ***type*** has size 10 bytes, then adding an integer 5 to a pointer to ***type*** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of `ptr2 - ptr1` would be 7.

The difference between two pointers has meaning only if both pointers point into the same array

When an integral value is added to or subtracted from a "pointer to ***type***," the result is also of type "pointer to ***type***."

There is no such element as "one past the last element," of course, but a pointer is allowed to assume such a value. If P points to the last array element, P + 1 is legal, but
P + 2 is undefined. If P points to one past the last array element, P - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a "pointer to one past the last element" leads to undefined behavior.

Informally, you can think of P + *n* as advancing the pointer by (*n* * **sizeof**(***type***)) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff_t* defined in stddef.h (**signed long** for **_ _huge** and **_ _far** pointers; **signed int** for all others). This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff_t*. In the expression *P1* - *P2*, where *P1* and *P2* are of type pointer to type (or pointer to qualified type), *P1* and *P2* must point to existing elements or to one past the last element. If *P1* points to the *i*-th element, and *P2* points to the *j*-th element, *P1* - *P2* has the value (*i* - *j*).

Pointer conversions

See also

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (***type****) will convert a pointer to type "pointer to ***type***."
See C++ specific for a discussion of C++ typecast mechanisms.

C++ reference declarations

See also

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See Referencing for complete details.

Arrays

See also

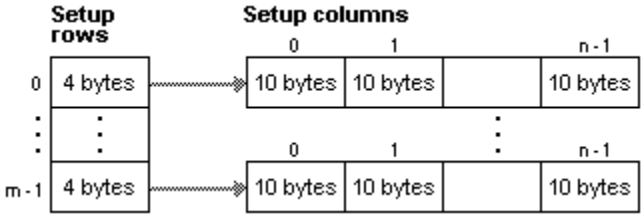
The declaration

type *declarator [<constant-expression>]*

declares an array composed of elements of ***type***. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implementation is for three rows and five columns but it can be very easily modified to accept run-time user input.




```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>

typedef long double TYPE;
typedef TYPE *OBJECT;
unsigned int rows = 3, columns = 5;

void de_allocate(OBJECT);

int main(VOID) {
  OBJECT matrix;
  unsigned int i, j;

  /* STEP 1: SET UP THE ROWS. */
  matrix = (OBJECT) calloc( rows, sizeof(TYPE *));

  /* STEP 2: SET UP THE COLUMNS. */
  for (i = 0; i < rows; ++i)
    matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));

    for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
        matrix[i][j] = i + j;    /* INITIALIZE */

  for (i = 0; i < rows; ++i) {
    printf("\n\n");
    for (j = 0; j < columns; ++j)
      printf("%5.2Lf", matrix[i][j]);
  de_allocate(matrix);
  return 0;
  }

void de_allocate(OBJECT x) {
  int i;
```

```
  for (i = 0; i < rows; i++)     /* STEP 1: DELETE THE COLUMNS */
    free(x[i]);

  free(x);     /* STEP 2: DELETE THE ROWS. */
  }
```

This code produces the following output:

```
0.00 1.00 2.00 3.00 4.00
1.00 2.00 3.00 4.00 5.00
2.00 3.00 4.00 5.00 6.00
```

Note: See Borland C++ Library Routines for a description of calloc, free, and printf.

In certain contexts, the first array declarator of a series might have no expression inside the brackets. Such an array is of indeterminate size. This is legitimate in contexts where the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array; neither does an array function parameter. As a special extension to ANSI C, Borland C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

Functions

See also

Functions are central to C and C++ programming. Languages such as Pascal distinguish between procedure and function. For C and C++, functions play both roles.

Declarations and definitions

See also

Each program must have a single external function named *main* marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see Linkage).

Functions are defined in your source files or made available by linking precompiled libraries.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide Borland C++ with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Note: In C++ you must always use function prototypes. We recommend that you also use them in C.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.

Declarations and prototypes

See also

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows

*<**type**> func*()

where ***type*** is the optional return type defaulting to **int**. In C++, this declaration means <***type***> *func*(**void**). A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

*<**type**> func*(*parameter-declarator-list*)*;*

Note: You can enable a warning within the IDE or with the command-line compiler: `"Function called without a prototype."`

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
   int limit = 32;
   char ch = 'A';
   long mval;
   mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for *lmax*, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to *lmax*. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to *lmax* would not match in size or content what *lmax* was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function *strcpy* takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```

In C++, *func*() also declares a function taking no arguments

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as *printf*), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Note: stdarg.h and varargs.h contain macros that you can use in user-defined functions with variable

numbers of parameters.

Here are some more examples of function declarators and prototypes:

```
int  f();              /* In C, a function returning an int with
            no information about parameters.
                 This is the K&R "classic style." */

int f();          /* In C++, a function taking no arguments */

int  f(void);                /* A function returning an int that takes
                  no parameters. */

int  p(int,long);            /* A function returning an int that
           accepts two parameters: the first,
            an int; the second, a long. */

int  _ _pascal q(void);      /* A pascal function returning
           an int that takes no parameters at all. */

char _ _far *s(char *source, int kind);   /*A function returning
                       a farpointer to a char
                       and accepting two parameters:
                       the first,a pointer to
                       a char;the second, an int. */

int  printf(char *format,...;  /*  A function returning an int and
                accepting a pointer to a char fixed
                parameter and any number of additional
                parameters of unknown type. */

int  (*fp)(int)              /* A pointer to a function returning an int
           and accepting a single int parameter. */
```

Definitions

See also

External function definitions gives the general syntax for external function definitions.

External function definitions

file

external-definition

file external-definition

external-definition:

function-definition

declaration

asm-statement

function-definition:

<declaration-specifiers> declarator <declaration-list>

compound-statement

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **_ _pascal**, **_ _cdecl**, **_ _export**, **_ _interrupt**, **_ _near**, **_ _far**, **_ _huge**, **_ _loadds**, **_ _saveregs**. The defaults depend on the memory model and compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is `func(void)`. The old style of *func* is legal in C but antiquated and possibly unsafe.
6. A function body representing the code to be executed when the function is called.

Note: You can mix elements from 1 and 2.

Formal parameter declarations

See also

The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) {                    // no args
int func(T1 t1, T2 t2, T3 t3=1) {   // three simple parameters, one
                                    // with default argument
int func(T1* ptr1, T2& tref) {      // A pointer and a reference arg
int func(register int i) {          // Request register for arg
int func(char *str,...) {           /* One string arg with a variable number
  of other
                                 args, or with a fixed number of args with var
  ying types */
```

In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, or enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all have automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal parameter declarators

Function calls and argument conversions

See also

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how Borland C++ deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, Borland C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard arithmetic conversions. When a function prototype is in scope, Borland C++ converts the given argument to the type of the declared parameter as if by assignment

When a function prototype includes an ellipsis (...), Borland C++ converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If your function prototype does not match the actual function definition, Borland C++ will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught.
C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

See also

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The Borland C++ structure type lets you handle complex data structures almost as easily as single variables. Structure initialization is discussed in Arrays, structures, and unions.

In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example

```
struct mystruct { ... }; // mystruct is the structure tag
    .
    .
    .
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged structures and typedefs

See also

If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];       // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, arry[20];
```

Usually, you don't need both a tag and a **typedef**: either can be used in structure declarations.

Untagged structure and union members are ignored during initialization.

Structure member declarations

See also

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in Borland C++ declaration syntax.

A structure member can be of any type, with two exceptions

- The member type cannot be the same as the **struct** type being currently declared:

```
struct mystruct { mystruct s } s1, s2; // illegal
```

  However, a member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct { mystruct *ps } s1, s2; // OK
```

  Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

- Except in C++, a member cannot have the type "function returning...," but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

Note: You can omit the **struct** keyword in C++.

Structures and functions

See also

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void);  // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s);         // directly
void func2(mystruct *sptr);     // via a pointer
void func3(mystruct &sref);     // as a reference (C++ only)
```

Structure member access

See also

Structure and union members are accessed using the following two selection operators:

- `.` (period)
- `->` (right arrow)

Suppose that the object *s* is of struct type *S*, and *sptr* is a pointer to *S*. Then if *m* is a member identifier of type *M* declared in *S*, the expressions *s.m* and *sptr->m* are of type *M*, and both represent the member object *m* in *S*. The expression *sptr->m* is a convenient synonym for `(*sptr).m`.

The operator . is called the direct member selector and the operator -> is called the indirect (or pointer) member selector. For example:

```
struct mystruct
{
   int i;
   char str[21];
   double d;
} s, *sptr = &s;
    .
    .
    .
s.i = 3;             // assign to the i member of mystruct s
sptr -> d = 1.23;    // assign to the d member of mystruct s
```

The expression *s.m* is an lvalue, provided that s is an lvalue and *m* is not an array type. The expression *sptr->m* is an lvalue unless *m* is an array type.

If structure *B* contains a field whose type is structure *A*, the members of *A* can be accessed by two applications of the member selectors

```
struct A {
   int j;
   double x;
};
struct B {
   int i;
   struct A a;
   double d;
} s, *sptr;
    .
    .
    .
s.i = 3;             // assign to the i member of B
s.a.j = 2;           // assign to the j member of A
sptr->d = 1.23;      // assign to the d member of B
(sptr->a).x = 3.14   // assign to x member of A
```

Each structure declaration introduces a unique structure type, so that in

```
struct A {
   int i,j;
   double d;
} a, a1;
struct B {
   int i,j;
   double d;
} b;
```

the objects *a* and *a1* are both of type struct *A*, but the objects *a* and *b* are of different structure types. Structures can be assigned only if the source and destination have the same type:

```
a = a1;   // OK: same type, so member by member assignment
a = b;    // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign member-by-member */
```

Structure word alignment

See also

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```
struct mystruct {
   int i;
   char str[21];
   double d;
} s;
```

the object s occupies sufficient memory to hold a 2-byte integer for a 16-bit program, or a 4-byte integer for a 32-bit program, a 21-byte string, and an 8-byte **double**. The format of this object in memory is determined by selecting the word alignment option. Without word alignment, s will be allocated 31 contiguous bytes (by the 16-bit compiler) or 33 contiguous bytes (by the 32-bit compiler).

Word alignment is off by default. If you turn on word alignment, Borland C++ pads the structure with bytes to ensure the structure is aligned as follows:

16-bit compiler alignment

1. The structure will start on a word boundary (even address)
2. Any non-**char** member will have an even byte offset from the start of the structure.
3. A final byte is added (if necessary) at the end to ensure that the whole structure contains an even number of bytes.

For the 16-bit compiler, with word alignment on, the structure would therefore have a byte added before the **double**, making a 32-byte object.

32-bit compiler alignment

1. The structure boundaries are defined by 4-byte multiples.
2. For any non-**char** member, the offset will be a multiple of the member size. A **short** will be at an offset that is some multiple of 2 **ints** from the start of the structure.
3. One to three bytes can be added (if necessary) at the end to ensure that the whole structure contains a 4-byte multiple.

For the 32bit compiler, with word alignment on, three bytes would be added before the **double**, making a 36-byte object.

Structure name spaces

See also

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example

```
goto s;
    .
    .
    .
s:            // Label
struct s {    // OK: tag and label name spaces different
   int s;     // OK: label, tag and member name spaces different
   float s;   // ILLEGAL: member name duplicated
} s;          // OK: var name space different. In C++, this can only
              // be done if s does not have a constructor.
union s {     // ILLEGAL: tag space duplicate
   int s;     // OK: new member space
   float f;
} f;          // OK: var name space
struct t {
   int s;     // OK: different member space
    .
    .
    .
} s;          // ILLEGAL: var name duplicate
```

Incomplete declarations

See also

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before *A* has been declared:

```
struct A;                   // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of *A* is called *incomplete* because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of *B* doesn't need the size of *A*.

Bit fields

See also

When you write an application for a 16-bit platform, you can declare **signed** or **unsigned** integer members as bit fields from 1 to 16 bits wide. For 32-bit platforms a bit field can be as much as 32 bits wide. You specify the bit-field width and optional identifier as follows:

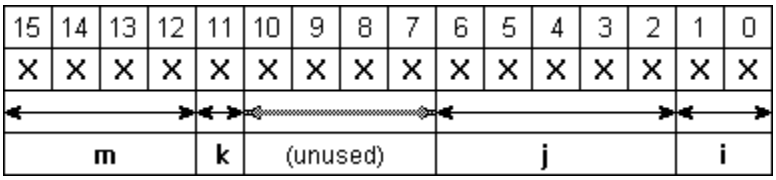
type-specifier <bitfield-id> : width;

where *type-specifier* is **char**, **unsigned char**, **int**, or **unsigned int**. Bit fields are allocated from low-order to high-order bits within a word. The expression *width* must be present and must evaluate to a constant integer in the range 1 to 32, depending on the target platform.

If the bit field identifier is omitted, the number of bits specified in *width* is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused. For example:

```
struct mystruct
  int       i : 2;
  unsigned  j : 5;
  int         : 4;
  int       k : 1;
  unsigned  m : 4;
) a, b, c;
```

produces the following layout:



Integer fields are stored in two's-complement form, with the leftmost bit being the MSB (most significant bit). With **int** (for example, **signed**) bit fields, the MSB is interpreted as a sign bit. A bit field of width 2 holding binary 11, therefore, would be interpreted as 3 if **unsigned**, but as -1 if **int**. In the previous example, the legal assignment `a.i = 6` would leave binary 10 = -2 in *a.i* with no warning. The signed **int** field *k* of width 1 can hold only the values -1 and 0, because the bit pattern 1 is interpreted as -1.

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (. and ->) used for non-bit-field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent

The expression *&mystruct.x* is illegal if *x* is a bit field identifier, because there is no guarantee that *mystruct.x* lies at a byte address

Unions

See also

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
   int i;
   double d;
   char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union** *myunion*, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time

Note: Unions correspond to the variant record types of Pascal and Modula-2.

sizeof(**union** *myunion*) and **sizeof**(*mu*) both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (. and ->), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); //OK: displays mu.d = 4.016
  printf("mu.i = %d\n",mu.i); //peculiar result
  mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); //OK: displays mu.ch = A
  printf("mu.d = %f\n",mu.d); //peculiar result
  muptr->i = 3;
printf("mu.i = %d\n",mu.i); //OK: displays mu.i = 3
```

The second *printf* is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous unions (C++ only)

See also

A union that doesn't have a tag and is not used to declare a named object (or other type) is called an *anonymous union*. It has the following form:

```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the `x.y` or `p->y` syntax.

Anonymous unions can't have member functions and at file level must be declared static. In other words, an anonymous union cannot have external linkage.

Union declarations

See also

The general declaration syntax for unions is similar to that for structures. The differences are

- Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (*Note that, because bit fields are machine dependent, they can pose problems when writing portable code.*)
- Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
- Unions can be initialized only through their first declared member:

```
union local87 {
   int i;
   double d;
   } a = { 20 };
```

- A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.

Enumerations

See also

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable *anyday* of this type, and a set of enumerators (*sun*, *mon*,...) with constant integer values

Borland C++ is free to store enumerators in a single byte when `Treat enums as ints is unchecked` (O|C|Code Generation) or the **-b** flag is used. The default is on (meaning **enums** are always **int**s) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, that is the type of each enumerator

In C, a variable of an enumerated type can be assigned any value of type **int**--no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;        // OK
anyday = 1;          // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```

In C++, you can omit the **enum** keyword if **days** is not the name of anything else in the same scope

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on). See Enumeration constants for more on enumeration constants

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* Initializer expression can include previously declared enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
             quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;         // OK
*daysptr = anyday;    // OK
mon = tues;           // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the

same name space as ordinary variable identifiers:

```
int mon = 11;
{
   enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
   /* enumerator mon hides outer declaration of int mon */
   struct days { int i, j;};   // ILLEGAL: days duplicate tag
   double sat;                 // ILLEGAL: redefinition of sat
}
mon = 12;                      // back in int mon scope
```

In C++, enumerators declared within a class are in the scope of that class.

In C++ it is possible to overload most operators for an enumeration. However, because the =, [ ], ( ), and -> operators must be overloaded as member functions, it is not possible to overload them for an **enum**. See the example on how to overload the postfix and prefix increment operators.

How to overload enum operators

```
// OVERLOAD THE POSTFIX AND PREFIX INCREMENT OPERATORS FOR enum
  #include <iostream.h>
  enum _SEASON { spring, summer, fall, winter };
  _SEASON operator++(_SEASON &s) {      // PREFIX INCREMENT
     _SEASON tmp = s; // SAVE THE ORIGINAL VALUE
     // DO MODULAR ARITHMETIC AND CAST THE RESULT TO _SEASON TYPE
     s = _SEASON( (s + 1) % 4 );        // INCREMENT THE ORIGINAL
     return s;                          // RETURN THE OLD VALUE
     }
  // UNNAMED int ARGUMENT IS NOT USED
  _SEASON operator++(_SEASON &s, int) { // POSTFIX INCREMENT
     _SEASON tmp = s;
     switch (s) {
        case spring: s = summer; break;
        case summer: s = fall; break;
        case fall:   s = winter; break;
        case winter: s = spring; break;
        }
     return (tmp);
     }
  int main(void) {
     _SEASON season = fall;
     cout << "\nThe season is " << season;
     cout << "\nIncrement the season: "<< ++season;
     cout << "\nNo change yet when using postfix: " << season++;
     cout << "\nFinally:" << season;
     return 0;
     }
```

This code produces the following output:

```
The season is 2
Increment the season: 3
No change yet when using postfix: 3
Finally:0
```

Assignment to enum types

See also

The rules for expressions involving **enum** types have been made stricter. The compiler enforces these rules with error messages if the compiler switch **-A** is turned on (which means strict ANSI C++).

Assigning an integer to a variable of **enum** type results in an error:

```
enum color
{
  red, green, blue
};

int f()
{
  color c;
  c = 0;
  return c;
}
```

The same applies when passing an integer as a parameter to a function. Notice that the result type of the expression `flag1|flag2` is **int**:

```
enum e
{
  flag1 = 0x01,
  flag2 = 0x02
};

void p(e);

void f()
{
  p(flag1|flag2);
}
```

To make the example compile, the expression `flag1|flag2` must be cast to the **enum** type: `e (flag1|flag2)`.

Expressions

See also

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Borland C++ expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Note: Borland C++ expressions shows how identifiers and operators are combined to form grammatically legal "phrases."

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands.The standard conversions are detailed in Methods used in standard arithmetic conversions. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by Borland C++ (see Evaluation order).

Expressions can produce an lvalue, an rvalue, or no value. Expressions might cause side effects whether they produce a value or not

The precedence and associativity of the operators are summarized in Associativity and precedence of Borland C++ operators. The grammar in Borland C++ expressions, completely defines the precedence and associativity of the operators

Borland C++ expressions

primary-expression:

- *literal*
- ***this*** (C++ specific)
- **::** *identifier* (C++ specific)
- **::** *operator-function-name* (C++ specific)
- **::** *qualified-name* (C++ specific)
- (*expression*)
- *name*

literal:

- *integer-constant*
- *character-constant*
- *floating-constant*
- *string-literal*

name:

- *identifier*
- *operator-function-name* (C++ specific)
- *conversion-function-name* (C++ specific)
- ~ *class-name* (C++ specific)
- *qualified-name* (C++ specific)

qualified-name: (C++ specific)

- *qualified-class-name* **::** *name*

postfix-expression:

- *primary-expression*
- *postfix-expression* [*expression*]
- *postfix-expression* (<*expression-list*>)
- *simple-type-name* (<*expression-list*>) (C++ specific)
- *postfix-expression* . *name*
- *postfix-expression* -> *name*

postfix-expression **++**

postfix-expression **--**

const_cast < *type-id* > (*expression*) (C++ specific)

dynamic_cast < *type-id* > (*expression*) (C++ specific)

reinterpret_cast < *type-id* > (*expression*) (C++ specific)

static_cast < *type-id* > (*expression*) (C++ specific)

typeid (*expression*) (C++ specific)

typeid (*type-name*) (C++ specific)

expression-list:

assignment-expression

expression-list **,** *assignment-expression*

unary-expression:

postfix-expression

++ *unary-expression*

- - *unary-expression*

unary-operator *cast-expression*

sizeof *unary-expression*

sizeof (*type-name*)

allocation-expression (C++ specific)

deallocation-expression (C++ specific)

unary-operator: one of **& * + - !**

allocation-expression: (C++ specific)

<**::**> **new** <*placement*> *new-type-name* <*initializer*>

<**::**> **new** <*placement*> (*type-name*) <*initializer*>

placement: (C++ specific)

(*expression-list*)

new-type-name: (C++ specific)

type-specifiers <*new-declarator*>

new-declarator: (C++ specific)

ptr-operator <*new-declarator*>

new-declarator [<*expression*>]

deallocation-expression: (C++ specific)

<**::**> **delete** *cast-expression*

<**::**> **delete [ ]** *cast-expression*

cast-expression:

unary-expression

(*type-name*) *cast-expression*

pm-expression:

cast-expression

pm-expression **.*** *cast-expression* (C++ specific)

pm-expression **->*** *cast-expression* (C++ specific)

multiplicative-expression:

pm-expression

multiplicative-expression ***** *pm-expression*

multiplicative-expression */* *pm-expression*

multiplicative-expression **%** *pm-expression*

additive-expression:

multiplicative-expression

additive-expression **+** *multiplicative-expression*

additive-expression **-** *multiplicative-expression*

shift-expression:

additive-expression

shift-expression **<<** *additive-expression*

shift-expression **>>** *additive-expression*

relational-expression:

shift-expression

relational-expression **<** *shift-expression*

relational-expression **>** *shift-expression*

relational-expression **<=** *shift-expression*

relational-expression **>=** *shift-expression*

equality-expression:

relational-expression

equality expression **==** *relational-expression*

equality expression **!=** *relational-expression*

AND-expression:

equality-expression

AND-expression **&** *equality-expression*

exclusive-OR-expression:

AND-expression

exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression **|** e*xclusive-OR-expression*

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression:

logical-AND-expression

logical-OR-expression **||** *logical-AND-expression*

conditional-expression:

logical-OR-expression

logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression:

conditional-expression

unary-expression *assignment-operator* *assignment-expression*

assignment-operator: one of

= ***=** **/=** **%=** **+=** **-=**

<< **=>** **>=** **&=** **^=** **|=**

expression:

assignment-expression

expression **,** *assignment-expression*

constant-expression:

conditional-expression

Expressions and C++

See also

C++ allows the overloading of certain standard C operators, as explained in Overloading Operator Functions. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator == might be defined in class *complex* to test the equality of two complex numbers without changing its normal usage with non-class data types.

An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the C language rules for operators and conversions might not apply to expressions in C++.

Evaluation order

See also

The order in which Borland C++ evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression

```
i = v[i++];  // i is undefined
```

The value of i depends on whether i is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for sum and total. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value

Borland C++ regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables *a*, *b*, *c*, and *f*, then the expression *f* = *a* + (*b* + *c*) forces (*b* + *c*) to be evaluated before adding the result to *a*.

Errors and overflows

See also

Associativity and precedence of Borland C++ operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, Borland C++ can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2*n* arithmetic on *n*-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.See _matherr and signal.

Equality operators

See also

There are two equality operators: **==** and **!=**. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

Note: Notice that **==** and **!=** have a lower precedence than the relational operators **<** and **>**, **<=**, and **>=**. Also, **==** and **!=** can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

equality-expression:

relational-expression

equality-expression **==** *relational-expression*

equality-expression **!=** *relational-expression*

Statements

See also

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. Borland C++ statements shows the syntax for statements.

Borland C++ statements

statement:
- *labeled-statement*
- *compound-statement*
- *expression-statement*
- *selection-statement*
- *iteration-statement*
- *jump-statement*
- *asm-statement*
- *declaration* (C++ specific)

labeled-statement:
- *identifier* : *statement*
- **case** *constant-expression* : *statement*
- **default** : *statement*

compound-statement:
- { <*declaration-list*> <*statement-list*> }

declaration-list:
- *declaration*
- *declaration-list declaration*

statement-list:
- *statement*
- *statement-list statement*

expression-statement:
- <*expression*> ;

asm-statement:
- **asm** *tokens newline*
- **asm** *tokens*;
- **asm** { *tokens*; <*tokens;*>= <*tokens;*>}

selection-statement:
- **if** (*expression*) *statement*
- **if** (*expression*) *statement* **else** *statement*
- **switch** (*expression*) *statement*

iteration-statement:
- **while** (*expression*) *statement*
- **do** *statement while* (*expression*) ;
- **for** (*for-init-statement* <*expression*> ; <*expression*>) *statement*

for-init-statement:
- *expression-statement*
- *declaration (C++ specific)*

jump-statement:
- **goto** *identifier* ;
- **continue** ;

break ;

return <*expression*> ;

Blocks

See also

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

Labeled statements

See also

A statement can be labeled in two ways:

- *label-identifier* **:** *statement*

  The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.

- **case** *constant-expression* **:** *statement*
  default : *statement*

  Case and default labeled statements are used only in conjunction with switch statements.

Expression statements

See also

Any expression followed by a semicolon forms an *expression statement*:

```
<expression>;
```

Borland C++ executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls

The *null statement* is a special case, consisting of a single semicolon (**;**). The null statement does nothing, and is therefore useful in situations where the Borland C++ syntax expects a statement but your program does not need one.

Selection statements

See also

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

Iteration statements

See also

Iteration statements let you loop a set of statements. There are three forms of iteration in Borland C++: **while**, **do while**, and **for** loops.

Jump statements

See also

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**

C++ specifics

See also

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function *func* declared twice in C with different argument types invokes a duplicated name error. Under C++, however, *func* will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This topic discusses the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

See C++ Exception Handling and C-Based Structured Exceptions for details on compiling C and C++ programs with exception handling.

Referencing

See also

While in C, you pass arguments only by value; in C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects and let you pass arguments to functions by reference. See the following topics for a discussion of referencing.

Simple references

Reference arguments

Reference/Indirect operators

Note: C++ specific pointer referencing and dereferencing is discussed in C++ specific operators.

Simple references

See also

The reference declarator can be used to declare references outside functions:

```
int  i  = 0;
int &ir = i;   // ir is an alias for i
ir = 2;        // same effect as i = 2
```

Note that type& var, type &var, and type & var are all equivalent.

This creates the lvalue *ir* as an alias for *i*, provided the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, `ir = 2` assigns 2 to *i*, and `&ir` returns the address of *i*.

Reference arguments

See also

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);     // ir is type "reference to int"
    .
    .
    .
int sum=3;
func1(sum);               // sum passed by value
func2(&sum);              // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by *func2*. On the other hand, *func1* gets a copy of the *sum* argument (passed by value), so sum itself cannot be altered by *func1*.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of x itself. Of course, the function can return a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The C method for changing *x* uses the actual argument `&x`, the address of *x*, rather than *x* itself. Although &x is passed by value, the function can access x through the copy of `&x` it receives. Even if the function does not need to change *x*, it is still useful (though subject to potentially dangerous side effects) to pass `&x`, especially if *x* is a large data structure. Passing *x* directly by value involves wasteful copying of the data structure.

Compare the three implementations of the function *treble*:

Implementation 1

```
int treble_1(int n)
{
   return 3 * n;
}
  .
  .
  .
int x, i = 4;
x = treble_1(i);          // x now = 12, i = 4
  .
  .
  .
```

Implementation 2

```
void treble_2(int* np)
{
   *np = (*np) * 3;
}
  .
  .
  .
treble_2(int& i);         // i now = 12
```

Implementation 3

```
void treble_3(int& n)     // n is a reference type
{
   n = 3 * n;
}
  .
```

```
   .
   .
treble_3(i);              // i now = 36
```

The formal argument declaration ***type***`&` t (or equivalently, ***type***`& t`) establishes *t* as type "reference to *type*." So, when *treble_3* is called with the real argument *i*, *i* is used to initialize the formal reference argument *n*. *n* therefore acts as an alias for *i*, so `n = 3*n` also assigns *3 * i* to *i*.

If the initializer is a constant or an object of a different type than the reference type, creates a temporary object for which the reference acts as an alias:

```
int& ir = 6;    /* temporary int object created, aliased by ir, gets value 6
  */
float f;
int& ir2 = f;   /* creates temporary int object aliased by ir2; f converted
                   before assignment */
ir2 = 2.0       // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

Classes

See also

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

class-key {<*distance-attrib*> <*distance-attrib*>} <*type-info*> *class-name*

<: *base-list*> { <*member-list*> };

class-key is one of **class**, **struct**, or **union**.

The optional *type-info* indicates a request for run-time type information about the class. You can compile with the **–RT** compiler option, or you can use the **_ _rtti** keyword. See the discussion of class typeinfo for more information.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class. The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes.

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that can affect which functions can access which members.

Class memory model specifications

See also

For 16-bit applications only, distance modifiers can be applied to a class declaration. The modifier(s) applied to a class declaration determine the addressing of the class's **this** pointer and the class's table of virtual functions (*vtable*). The distance modifiers allowed for class declarations, and their effect on the addressing of **this** and the vtable are as follows:

Class memory model specifications

Modifier	*this	vtable
_ _near	near	near
_ _far	far	near
_ _huge	far	far
_ _huge _ _near	near	far
_ _export	far	far
_ _import	far	far

If you're importing classes that are declared with the modifier **_ _huge**, you must change the modifier to the keyword **_ _import**. The **_ _huge** modifier merely causes far addressing of the virtual tables (the same effect as the **–Vf** compiler option). The **_ _import** modifier makes all function and static addresses default to far

See Exporting and Importing Classes for a discussion of declaration of classes used in DLLs.

Class names

See also

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted. See Untagged structures and typedefs for discussion of untagged structures.

Class types

See also

The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```
class X { ... };
X x, &xr, *xptr, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X */
struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// C would have
// struct Y y, *yptr, yarray[10];
union Z { ... };
Z z, &zr, *zptr, zarray[10];
// C would have
// union Z z, *zptr, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++, they are needed only when the class names, Y and Z, are hidden (see Class name scope)

Class name scope

See also

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union**, must be used with the class name. For example,

```
struct S { ... };
int S(struct S *Sptr);
void func(void) {
   S t;         // ILLEGAL declaration: no class key and function S in scope
   struct S s;  // OK: elaborated with class key
   S(&s);       // OK: this is a function call
}
```

C++ also allows an incomplete class declaration:

```
class X;  // no members, yet!
```

Incomplete declarations permit certain references to class name *X* (usually references to pointers to class objects) before the class has been fully defined. See Structure member declarations for more information. Of course, you must make a complete class declaration with members before you can define and use class objects.

Class objects

See also

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class.

Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class member list

See also

The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes), function declarations, and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions

See also

A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

The keyword this

See also

Nonstatic member functions operate on the class type object they are called with. For example, if *x* is an object of class *X* and *f()* is a member function of *X*, the function call `x.f()` operates on *x*. Similarly, if *xptr* is a pointer to an *X* object, the function call `xptr->f()` operates on **xptr*. But how does *f* know which instance of *X* it is operating on? C++ provides *f* with a pointer to *x* called **this**. **this** is passed as a hidden argument in all calls to nonstatic member functions.

this is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If *x.f(y)* is called, for example, where *y* is a member of *X*, **this** is set to *&x* and *y* is set to **this->***y*, which is equivalent to *x.y*.

Inline functions

See also

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline function*.

Borland C++ can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an inline expansion.

Note: The Borland C++ compiler can ignore requests for inline expansion.

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of *func*:

```
int i;                          // global int
class X {
public:
   char* func(void) { return i; }  // inline by default
   char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit **inline** specifier. The *i* returned by *func* is the **char*** *i* of class *X* (see Member scope).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by Borland C++. For example,

```
inline void f1() throw(int)
   {
   // Warning: Functions with exception specifications are not expanded inli
  ne
   }
```

The remaining restrictions apply only when destructor cleanup is enabled.

Note: Destructors are called by default. See Setting Exception Handling Options for information about exception-handling switches.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
   foo();
   ~foo();
   };
inline void f2(foo& x) {
   // no warning, f2() can be expanded inline
   }
inline void f3(foo x) {
   // Warning: Functions taking class-by-value argument(s) are
   //          not expanded inline in function f3(foo)
   }
```

An inline function that returns a class with a destructor by value will not be expanded inline whenever

there are variables or temporaries that need to be destructed within the return expression:

```
struct foo {
   foo();
   ~foo();
   };
inline foo f4() {
   return foo();
   // no warning, f4() can be expanded inline
   }
inline foo f5() {
   foo X;
   return foo(); // Object X needs to be destructed
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f5()
   }
inline foo f6() {
   return ( foo(), foo() );  // temporary in return value
   // Warning:   Functions containing some return statements are
   //            not expanded inline in function f6()
   }
```

Static members

See also

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy "exists" for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If *x* is a static member of class *X*, it can be referenced as *X::x* (even if objects of class *X* haven't been created yet). It is still possible to access x using the normal member access operators. For example, *y.x* and *yptr->x*, where *y* is an object of class *X* and *yptr* is a pointer to an object of class *X*, although the expressions *y* and *yptr* are not evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};
void g(void); {
   X obj;
   func(1, &obj);       // error unless there is a global func()
                        // defined elsewhere
   X::func(1, &obj);    // calls the static func() in X
                        // OK for static functions only
   obj.func(1, &obj);   // so does this (OK for static and
                        // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don't have a **this** pointer, and therefore cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, *func* might be defined as follows:

```
void X::func(int i, X* ptr)
{
   member_int = i;        // which object does member_int
                          // refer to? Error
   ptr->member_int = i;   // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```
class X {
   static int x;
   class inner {
      static float f;
      void func(void);      // nested declaration
      };
};
int X::x = 1;
float X::inner::f = 3.14;  // initialization of nested static
X::inner::func(void) {     /*  define the nested function */  }
```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

Member scope

See also

The expression `X::func()` in the example in Inline functions and exceptions uses the class name *X* with the scope access modifier to signify that *func*, although defined "outside" the class, is indeed a member function of *X* and exists within the scope of *X*. The influence of *X*:: extends into the body of the definition. This explains why the *i* returned by *func* refers to *X::i*, the **char*** *i* of *X*, rather than the global **int** *i*. Without the *X*:: modifier, the function func would represent an ordinary non-class function, returning the global **int** *i*.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class *X* can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see The keyword this). For example:

```
class X {
public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right);   // define elsewhere
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr->i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class *X*, the expression *X::m* is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. It is important to note that, even if the class name *X* is hidden by a non-type name, the qualified name *X::m* will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class *X* cannot contain objects of class *X*, but can contain pointers or references to objects of class *X* (note the similarity with C's structure and union types).

Nested types

See also

Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the ***xxx::yyy*** notation, except when in the scope of the appropriate class.

A class declared within another class is called a *nested class*. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
   typedef int t;  // 'outer::t' is a typedef name
   struct inner    // 'outer::inner' is a class
   {
      static int x;
   };
   static int x;
      int f();
   class deep;     // nested declaration
};
int outer::x;     // define static data member
int outer::f() {
   t x;           // 't' visible directly here
   return x;
   }
int outer::inner::x;    // define static data member
outer::t x;             //  have to use 'outer::t' here
class outer::deep { };  // define the nested class here
```

With Borland C++ 2.0, any tags or **typedef** names declared inside a class actually belong to the global (file) scope. For example:

```
struct foo
{
   enum bar { x };   // 2.0 rules: 'bar' belongs to file scope
                     // 2.1 rules: 'bar' belongs to 'foo' scope
};
bar x;
```

The preceding fragment compiles without errors. But because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member access control

See also

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

- **public**: The member can be used by any function.
- **private**: The member can be used only by member functions and friends of the class it's declared in.
- **protected**: Same as for **private**. Additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in Base and derived class access.)

Note: Friend function declarations are not affected by access specifiers (see Friends of classes for more information).

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
   int i;    // X::i is private by default
   char ch;  // so is X::ch
public:
   int j;    // next two are public
   int k;
protected:
   int l;    // X::l is protected
};
struct Y {
   int i;    // Y::i is public by default
private:
   int j;    // Y::j is private
public:
   int k;    // Y::k is public
};
union Z {
   int i;    // public by default; no other choice
   double d;
};
```

Note: The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

Base and derived class access

See also

When you declare a derived class *D*, you list the base classes *B1, B2*, ... in a comma-delimited *base-list*:

class-key D : base-list { <member-list> }

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) *D* can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by *D*? *D* might want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

Note: Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basest of the base classes, those that do not inherit.

When declaring *D*, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the *base-list*:

```
class D : public B1, private B2, ... {
   .
   .
   .
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they can alter the access attributes of base members as viewed by the derived class.

The default is private if D is a class declaration, and public if D is a struct declaration.

Note: Unions cannot have base classes, and unions cannot be used as base classes.

The derived class inherits access attributes from a base class as follows:

- **public** base class: **public** members of the base class are **public** members of the derived class. **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **protected** base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **private** members of the base class remain **private** to the base class.

Note that **private** members of a base class are always inaccessible to member functions of the derived class *unless* **friend** declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {               // default for class is private A
   .
   .
   .
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */
class Y : B, public C {     // override default for C
   .
   .
   .
}
/* struct S is derived from D */
struct S : D {              // default for struct is public D
   .
   .
   .
}
```

```
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E {   // override default for D
                            // E is public by default
  .
  .
  .
}
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations of the derived class. For example:

```
class B {
   int a;               // private by default
public:
   int b, c;
   int Bfunc(void);
};
class X : private B {   // a, b, c, Bfunc are now private in X
   int d;               // private by default, NOTE: a is not
                        // accessible in X
public:
   B::c;                // c was private, now is public
   int e;
   int Xfunc(void);
};
int Efunc(X& x);        // external to B and X
```

The function *Efunc()* can use only the public names *c*, *e*, and *Xfunc()*.

The function *Xfunc()* is in *X*, which is derived from **private** *B*, so it has access to

- The “adjusted-to-public” *c*
- The “private-to-*X*” members from *B*: *b* and *Bfunc()*
- X’s own private and public members: *d*, *e*, and *Xfunc()*

However, *Xfunc()* cannot access the “private-to-*B*” member, *a*.

Virtual base classes

See also

A **virtual** class is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ...};
class D : B, B { ... };  // ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }  // OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword **virtual** to the base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y, virtual public Z
   X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // non-virtual base class
X();   // derived class
```

Friends of classes

See also

A **friend** *F* of a class *X* is a function or class, although not a member function of *X*, with full access rights to the private and protected members of *X*. In all other respects, *F* is a normal function with respect to scope, declarations, and definitions.

Since *F* is not a member of *X*, it is not in the scope of *X*, and it cannot be called with the *x.F* and *xptr->F* selector operators (where x is an X object and xptr is a pointer to an X object).

If the specifier **friend** is used with a function declaration or definition within the class *X*, it becomes a friend of *X*.

friend functions defined within a class obey the same inline rules as member functions (see Inline functions). **friend** functions are not affected by their position within the class or by any access specifiers. For example:

```
class X {
   int i;                            // private to X
   friend void friend_func(X*, int);
/* friend_func is not private, even though it's declared in the private sect
  ion */
public:
     void member_func(int);
};
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;
/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class Y into friends of class X with a single declaration:

```
class Y;                       // incomplete declaration
class X {
   friend Y;
   int i;
   void member_funcX();
};
class Y; {                     // complete the declaration
   void friend_X1(X&);
   void friend_X2(X*);
        .
        .
        .
};
```

The functions declared in *Y* are friends of *X*, although they have no **friend** specifiers. They can access the private members of *X*, such as *i* and *member_funcX*.

It is also possible for an individual member function of class *X* to be a friend of class *Y*:

```
class X {
        .
        .
        .
   void member_funcX();
}
class Y {
```

```
    int i;
    friend void X::member_funcX();
          .
          .
          .
};
```

Class friendship is not transitive: *X* friend of *Y* and *Y* friend of *Z* does not imply *X* friend of *Z*. Friendship is not inherited.

Introduction to constructors and destructors

See also

There are several special member functions that determine how the objects of a class are created, initalized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even **void**).
- They cannot be inherited, though a derived class can call the base class's constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be **virtual**, but constructors cannot. (See Virtual destructors.)
- You can't take their addresses.

```
int main (void)
{
   .
   .
   .
    void *ptr = base::base;     // illegal
   .
   .
   .
}
```

- Constructors and destructors can be generated by Borland C++ if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
   .
   .
   .
   X *p;
   .
   .
   .
   p->X::~X();                 // legal call of destructor
   X::X();                     // illegal call of constructor
   .
   .
   .
}
```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class *X* to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class *X*. Note that this rule applies only to any constructor with *one* parameter and no initializers that use the "**=**" syntax.

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                    // illegal: Y(X(1)) not tried
```

If **class** *X* has one or more constructors, one of them is invoked each time you define an object *x* of **class** *X*. The constructor creates x and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

See also

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the *main* function is called. When the **#pragma startup** directive is used to install a function prior to the *main* function, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
   X();   // class X constructor
};
```

A **class** *X* constructor cannot take *X* as an argument:

```
class X {
public:
   X(X);                    // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor . A constructor that accepts no parameters is called the default constructor .

Constructor defaults

See also

The default constructor for **class** *X* is one that takes no arguments; it usually has the form `X::X()`. If no user-defined constructors exist for a class, Borland C++ generates a default constructor. On a declaration such as *X x*, the default constructor creates the object *x*.

Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes no arguments and must not be confused with, say, `X::X(int = 0)`, which can be called with *no* arguments as a default constructor, or can take an argument.

You should avoid ambiguity in calling constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};
int main() {
   X one(10);  // OK; uses X::X(int)
   X two;      // illegal; ambiguous whether to call X::X() or
               // X::X(int = 0)
   return 0;
}
```

The copy constructor

See also

A copy constructor for **class** *X* is one that can be called with a single argument of type *X* as follows:

```
X::X(X&)
```

or

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

Borland C++ generates a copy constructor for **class** *X* if one is needed and no other constructor has been defined in **class** *X*. The copy constructor that is generated by the Borland C++ compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor if your program creates aggregate, complex types such as **class**, **struct**, and arrays The copy constructor is also called when you pass a class argument by value to a function.

See also the discussion of member-by-member class assignment. You should define the copy constructor if you overload the assignment operator.

Overloading constructors

See also

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X {
   int    integer_part;
   double double_part;
public:
   X(int i)    { integer_part = i; }
   X(double d) { double_part = d; }
};
int main() {
   X one(10);   // invokes X::X(int) and sets integer_part to 10
   X one(3.14); // invokes X::X(double) setting double_part to 3.14
   return 0;
}
```

Order of calling constructors

See also

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y();   // base class constructor
X();   // derived class constructor
```

For the case of multiple base classes,

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y();  // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code:

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // nonvirtual base class
X();   // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy
             // base is constructed only once
base2();     // nonvirtual base of virtual base level2
             // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence

of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class initialization

See also

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
   int i;
public:
   X();          // function bodies omitted for clarity
   X(int x);
   X(const X&);
};
void main()
{
   X one;         // default constructor invoked
   X two(1);      // constructor X::X(int) is used
   X three = 1;   // calls X::X(int)
   X four = one; // invokes X::X(const X&) for copy
   X five(two);   // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways:

- It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

- An initializer list can be used prior to the function body:

```
class X
{
   int a, b, &c;  // Note the reference variable.
public:
   X(int i, int j) : a(i), b(j), c(a) {}
};
```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to *x::a* and 2 to *x::b*. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Note: Base class constructors must be declared as either **public** or **protected** to be called from a derived class.

```
class base1
{
   int x;
public:
```

```
   base1(int i) { x = i; }
};

class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};
class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};
```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of *base1* with the value 5 and *base2* with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
  int a, b;
public:
  X(int i, j) :  a(i), b(a+j) {}
};
```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to *x::a* and 2 to *x::b*.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
};
class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized 'a'
};
```

With this class setup, a call of `derived d(1)` will *not* result in a value of 10 for the base class member *x*. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
  .
  .
  .
}
```

Destructors

See also

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
   ~X(){};  // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

Invoking destructors

See also

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).

atexit, #pragma exit, and destructors

See also

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the *main* function, are destroyed as they go out of scope. The order of execution at the end of a Borland C++ program is as follows:

- *atexit()* functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

exit and destructors

See also

When you call *exit* from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

abort and destructors

See also

If you call *abort* anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**

```
#include <stdlib.h>
class X {
public:
   .
   .
   .
    ~X(){};
   .
   .
   .
};
void* operator new(size_t size, void *ptr)
{
   return ptr;
}
char buffer[sizeof(X)];
void main() {
   X* pointer = new X;
   X* exact_pointer;
   exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
        .
        .
        .
   delete pointer;                 // delete used to destroy pointer
   exact_pointer->X::~X();         // direct call used to deallocate
}
```

Virtual destructors

See also

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
/* How virtual affects the order of destructor calls.
   Without a virtual destructor in the base class, the derived
   class destructor won't be called. */
#include <iostream.h>
class color {
public:
   virtual ~color() {  // Virtual destructor
      cout << "color dtor\n";
      }
};
class red : public color {
public:
   ~red() {  // This destructor is also virtual
      cout << "red dtor\n";
      }
};
class brightred : public red {
public:
   ~brightred() {  // This destructor is also virtual
       cout << "brightred dtor\n";
       }
};
int main() {
   color *palette[3];
   palette[0] = new red;
   palette[1] = new brightred;
   palette[2] = new color;

   // The destructors for red and color are called.
   delete palette[0];
   cout << endl;

   // The destructors for bright red, red, and color are called.
   delete palette[1];
   cout << endl;

   // The destructor for color is called.
   delete palette[2];
   return 0;
}
```

Program Output:

```
red dtor
color dtor

brightred dtor
red dtor
color dtor

color dtor
```

However, if no destructors are declared as virtual, **delete** *palette*[0], **delete** *palette*[1], and **delete** *palette*[2] would all call only the destructor for class *color*. This would incorrectly destruct the first two elements, which were actually of type *red* and *brightred*.

Polymorphic classes

See also

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.

Virtual functions

See also

virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a *pure* function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

```
virtual int funct1(void);       // A virtual function declaration.
virtual int funct2(void) = 0;   // A pure function declaration.
virtual void funct3(void) = 0 { // This is a valid declaration.
   // Some code here.
   };
```

Note: See Abstract classes for a discussion of pure virtual functions.

When you declare **virtual** functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a **friend** of another class.
- They cannot be a static member.

A **virtual** function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a **virtual** function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined **virtual** functions differing only in return type is discussed below.) A redefined function is said to *override* the base class function.

You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not virtual. In such a case, `int Derived::Fun(int)` is said to hide any other versions of `Fun(int)` that exist in any base classes. In addition, if class *Derived* defines other versions of *Fun()*, (that is, versions of *Fun()* with different signatures) such versions are said to be overloaded versions of *Fun()*.

Virtual function return types

Generally, when redefining a **virtual** function, you cannot change just the function return type. To redefine a **virtual** function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the **virtual** function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when *both* of the following conditions are met:

- The overridden **virtual** function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class *B* and class *D* (derived publicly from *B*) each contain a **virtual** function *vf*, then if *vf* is called for an object *d* of *D*, the call made is `D::vf()`, even when the access is via a pointer or reference to *B*. For example,

```
struct X {};      // Base class.
struct Y : X {};       // Derived class.
struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();    // Return type is a pointer to base. This can
    //  be overridden.
   };
```

```
class D : public B {
public:
   virtual void vf1(); // Virtual specifier is legal but redundant.
   void vf2(int);      // Not virtual, since it's using a different
    //  arg list. This hides B::vf2().
// char vf3();         // Illegal: return-type-only change!
   void f();
   Y*   pf();          // Overriding function differs only
    //  in return type. Returns a pointer to
    //  the derived class.
   };
void extf() {
   D d;  // Instantiate D
   B* bp = &d; // Standard conversion from D* to B*
    // Initialize bp with the table of functions
       // provided for object d. If there is no entry for a
    // function in the d-table, use the function
       //  in the B-table.
   bp->vf1();  // Calls D::vf1
   bp->vf2();  // Calls B::vf2 since D's vf2 has different args
   bp->f();    // Calls B::f (not virtual)
   X* xptr = bp->pf();   // Calls D::pf() and converts the result
    //  to a pointer to X.
   D* dptr = &d;
   Y* yptr = dptr->pf(); // Calls D::pf() and initializes yptr.
    //  No further conversion is done.
   }
```

The overriding function *vf1* in *D* is automatically **virtual**. The **virtual** specifier *can* be used with an overriding function declaration in the derived class. If other classes will be derived from *D*, the **virtual** keyword is required. If no further classes will be derived from *D*, the use of **virtual** is redundant.

The interpretation of a **virtual** function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

Abstract classes

See also

An abstract class is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {       // abstract class
   point center;
        .
        .
        .
public:
   where() { return center; }
   move(point p) { center = p; draw(); }
   virtual void rotate(int) = 0; // pure virtual function
   virtual void draw() = 0;      // pure virtual function
   virtual void hilite() = 0;    // pure virtual function
        .
        .
        .
}
shape x; // ERROR: attempt to create an object of an abstract class
   shape* sptr;   // pointer to abstract class is OK
   shape f();     // ERROR: abstract class cannot be a return type
int g(shape s);   // ERROR: abstract class cannot be a function argument typ
  e
shape& h(shape&); // reference to abstract class as return
    // value or function argument is OK
```

Suppose that *D* is a derived class with the abstract class *B* as its immediate base class. Then for each pure virtual function pvf in *B*, if *D* doesn't provide a definition for *pvf*, *pvf* becomes a pure member function of *D*, and *D* will also be an abstract class.

For example, using the class *shape* previously outlined,

```
class circle : public shape { // circle derived from abstract class
   int radius;                // private
public:
   void rotate(int) { }       // virtual function defined: no action
                              //  to rotate a circle
   void draw();               // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

C++ scope

See also

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

Class scope

See also

The name *M* of a member of a class *X* has class scope “local to *X*”; it can be used only in the following situations:

- In member functions of *X*
- In expressions such as `x.M`, where *x* is an object of *X*
- In expressions such as *xptr*`->`*M*, where *xptr* is a pointer to an object of *X*
- In expressions such as `X::M` or `D::M`, where *D* is a derived class of *X*
- In forward references within the class of which it is a member

Names of functions declared as friends of *X* are not members of *X*; their names simply have enclosing scope.

Hiding

See also

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: `X::M`. A hidden file scope (global) name can be referenced with the unary operator :: (for example, *::g*). A class name *X* can be hidden by the name of an object, function, or enumerator declared within the scope of *X*, regardless of the order in which the names are declared. However, the hidden class name *X* can still be accessed by prefixing *X* with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name x is immediately after its complete declaration but before its initializer, if one exists.

C++ scoping rules summary

See also

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, *and* if the name is not qualified by the binary :: operator or the member selection operators **.** and **->**, then the name must be a global object, function, or enumerator.
- If the name *n* appears in any of the forms *X::n, x.n* (where *x* is an object of *X* or a reference to X), or *ptr->n* (where *ptr* is a pointer to *X*), then *n* is the name of a member of X or the member of a class from which *X* is derived.
- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class *X* must either be declared in the block it occurs in or in an enclosing block, be a member of class *X* or a base class of *X*, or be a global name. The declaration of a local name *n* hides declarations of n in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer (see *ctor-initializer* in the class declarator syntax in Borland C++ declaration syntax,) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Preprocessor Directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The Borland C++ preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. Borland C++ supports these preprocessor directives:

# (null directive)	#ifdef
#define	#ifndef
#elif	#include
#else	#line
#endif	#pragma
#error	#undef
#if	

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

(null directive)

Directives

Syntax

#

Description

The null directive consists of a line containing the single character #. This line is always ignored.

#define

See also Example Directives

Syntax

```
#define macro_identifier <token_sequence>
```

Description

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Each occurrence of *macro_identifier* in your source code following this control line will be replaced in the original position with the possibly empty *token_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

An empty token sequence results in the removal of each affected macro identifier from the source code.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor. There are these restrictions to macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro won't be expanded during its own expansion (so `#define A A` won't expand indefinitely).

Example

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
#define GETSTD #include <stdio.h>
```

#error

Example Directives

Syntax

```
#error errmsg
```

Description

The **#error** directive generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional statement that is true for the undesired case.

Example

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

#if, #elif, #else, and #endif

See also Directives

Syntax

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
  .
  .
  .
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```

Description

Borland C++ supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the Borland C++ compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with *next-section*. In the *false case*, control passes to the next **#elif** line (if any) where constant-expression-2 is evaluated. If true, *section-2* is processed, after which control moves on to the matching **#endif**. Otherwise, if *constant-expression-2* is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be matched with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

#ifdef and #ifndef

See also Directives

Syntax

```
#ifdef identifier
#ifndef identifier
```

Description

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous #define command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

#ifndef tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the #if, #elif, #else, and #endif.

An identifier defined as NULL is considered to be defined.

#include

Example Directives

Syntax

```
#include <header_name>
#include "header_name"
#include macro_identifier
```

Description

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, *header_name* is never scanned for macro identifiers. *header_name* must be a valid DOS file name with an extension (traditionally .h for header) and optional path name and path delimiters.
- The third version assumes that neither **<** nor " appears as the first non-whitespace character following **#include**; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <*header_name*> or "*header_name*" formats.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header_name*, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file.

Example

This **#include** statement causes it to look for stdio.h in the standard include directory.

```
#include <stdio.h>
```

This **#include** statement causes it to look for MYINCLUD.H in the current directory, then in the default directories.

```
#include "myinclud.h"
```

After expansion, this **#include** statement causes the preprocessor to look in C:\BC5\INCLUDE\ MYSTUFF.H and nowhere else.

```
#define myinclud "C:\BC5\INCLUDE\MYSTUFF.H"
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\BC5\INCLUDE\\MYSTUFF.H" */
#include  myinclud
/* macro expansion */
```

#line

Directives

Syntax

```
#line integer_constant  <"filename">
```

Description

You can use the **#line** directive to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program.

The **#line** directive indicates that the following source line originally came from line number *integer_constant* of *filename*. Once the *filename* has been registered, subsequent **#line** commands relating to that file can omit the explicit *filename* argument.

Macros are expanded in **#line** arguments as they are in the #include directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

#pragma summary

Directives

Syntax

```
#pragma directive-name
```

Description

With **#pragma**, Borland C++ can define the directives it wants without interfering with other compilers that support **#pragma**. If the compiler doesn't recognize *directive-name*, it ignores the **#pragma** directive without any error or warning message.

Borland C++ supports the following **#pragma** directives:

#pragma argsused
#pragma anon_struct
#pragma codeseg
#pragma comment
#pragma exit
#pragma hdrfile
#pragma hdrstop
#pragma inline
#pragma intrinsic
#pragma message
#pragma option
#pragma saveregs
#pragma startup
#pragma warn

#pragma argsused

See also #pragma

Syntax

```
#pragma argused
```

Description

The **argsused** pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

#pragma anon_struct

See also #pragma

Syntax

```
#pragma anon_struct on
#pragma anon_struct off
```

Description

The **anon_struct** directive allows you to compile anonymous structures embedded in classes.

```
#pragma anon_struct on
struct S {
   int i;
   struct {  // Embedded anonymous struct
      int   j ;
      float x ;
   };
   class {  // Embedded anonymous class
   public:
      long double ld;
   };
S() { i = 1; j = 2; x = 3.3; ld = 12345.5;}
};
#pragma anon_struct off

void main() {
   S mystruct;
   mystruct.x = 1.2;  // Assign to embedded data.
   }
```

#pragma codeseg

See also #pragma

Syntax

```
#pragma codeseg <seg_name> <"seg_class"> <group>
```

Description

The **codeseg** directive lets you name the segment, class, or group where functions are allocated. If the pragma is used without any of its options arguments, the default code segment is used for function allocation.

#pragma comment

See also #pragma

Syntax

```
#pragma comment (comment type, "string")
```

Description

The **comment** directive lets you write a comment record into an output file. The *comment type* can be one of the following values:

Value	Explanation
exestr	The linker writes *string* into an .OBJ file. Your specified *string* is placed in the executable file. Such a string is never loaded into memory but can be found in the executable file by use of a suitable file search utility.
lib	Writes a comment record into an .OBJ file. The comment record is used by the linker as a library-search directory. A library module that is not specified in the linker's response-file can be specified by the **comment** LIB directive. The linker includes the library module name specified in *string* as the last library. Multiple modules can be named and linked in the order in which they are named.
user	The compiler writes *string* into the .OBJ file. The specified string is ignored by the linker.

#pragma exit and **#pragma startup**

See also #pragma

Syntax

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

Description

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the *main* function is called), or program exit (just before the program terminates through **_exit**).

The specified *function-name* must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as:

```
void func(void);
```

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100.

Note: Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.

#pragma hdrfile

See also #pragma

Syntax

```
#pragma hdrfile "filename.CSM"
```

Description

This directive sets the name of the file in which to store precompiled headers.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option -H=filename or Use Precompiled Headers to change the name of the file used to store precompiled headers.

#pragma hdrstop

See also #pragma

Syntax

```
#pragma hdrstop
```

Description

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers.

#pragma inline

See also #pragma

Syntax

```
#pragma inline
```

Description

This directive is equivalent to the **-B** command-line compiler option or the IDE inline option.

This is best placed at the top of the file, because the compiler restarts itself with the **-B** option when it encounters **#pragma inline**.

#pragma intrinsic

See also Example #pragma

Syntax

```
#pragma intrinsic [-]function-name
```

Description

Use **#pragma intrinsic** to override command-line switches or IDE options to control the inlining of functions.

When inlining an intrinsic function, always include a prototype for that function before using it.

Example

This example causes the compiler to generate code for *strcpy* in your function:

```
#pragma intrinsic strcpy
```

while this version prevents the compiler from inlining *strcpy*:

```
#pragma intrinsic -strcpy
```

#pragma message

See also Example #pragma

Syntax

```
#pragma message ("text" ["text"["text" ...]])
#pragma message text
```

Description

Use **#pragma message** to specify a user-defined message within your program code.

The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses. (This form is compatible with MSC.) The second form uses the text following the **#pragma** for the text of the warning message. With both forms of the **#pragma**, any macro references are expanded before the message is displayed.

Display of user-defined messages is on by default and can be turned on or off with the User-Defined Warnings (Options|Project|Messages|General) in the IDE. This option corresponds to the 16/32 bit compiler's **-wmsg** switch.

Example

The following example displays either "You are compiling using version xxx of BC++" (where xxx is the version number) or "Sorry, you are not using the Borland C++ compiler".

```
#ifdef __BORLANDC__
#pragma message You are compiling using version __BORLANDC__ of BC++.
#else
#pragma message ("Sorry, you are not using the Borland C++ compiler")
#endif
```

#pragma option

See also #pragma

Syntax

```
#pragma option [options...]
```

Description

Use **#pragma option** to include command-line options within your program code.

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include:

-B -c -d*name*
-D*name=string* -e*filename* -E
-Fx -h -l*filename*
-lexset -M -o
-P -Q -S
-T -U*name* -V
-X -Y

You can use **#pragmas**, **#includes**, **#define**, and some **#ifs** in the following cases:

- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear only in a **#pragma option** command before these events. These options are:

-E*filename* -f -i#
-m* -n*path* **-o*filename***
-u -W -z
*

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

-1 -h -r
-2 -k -rd
-a -N -v
-ff -O -y
-G -p -Z

The following options can be changed at any time and take effect immediately:

-A -g*n* -zE
-b -j*n* -zF
-C -K -zH
-d -w*xxx*

The options can appear followed by a dot (.) to reset the option to its command-line state.

#pragma saveregs

See also #pragma

Syntax

```
#pragma saveregs
```

Description

The **saveregs** pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

#pragma warn

See also Example #pragma

Syntax

```
#pragma warn [+|-|.]www
```

Description

The **warn** pragma lets you override specific **-w*xx*** command-line options or check Display Warningsl in the Messages options.

Example

If your source code contains the directives:

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned on, the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began. See the command-line options summary for a complete list of the three-letter abbreviations and the warnings to which they apply.

#undef

See also Example Directives

Syntax

```
#undef macro_identifier
```

Description

You can undefine a macro using the **#undef** directive. **#undef** detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined. No macro expansion occurs within **#undef** lines.

The state of being *defined* or *undefined* turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is *exactly* the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Example

```
#define BLOCK_SIZE 512
  .
  .
  .
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
  .
  .
  .
#define BLOCK_SIZE 128   /* redefinition */
```

Using the -D and -U command-line options

See also

Identifiers can be defined and undefined using the command-line compiler options -D and -U.

The command line

```
BCC32 -Ddebug=1; paradox=0; X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
#define X
#undef mysym
```

in the program.

Keywords and Protected Words as Macros

It is legal but ill-advised to use Borland C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers *cannot* appear immediately following a #define or #undef directive:

__DATE__ __FILE__ __LINE__

__STDC__ __TIME__

Macros with Parameters

See also

The following syntax is used to define a macro with parameters:

`#define macro_identifier(<arg_list>) token_sequence`

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the **(**. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *placeholder*.

Such macros are called by writing

`macro_identifier<whitespace>(<actual_arg_list>)`

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the #define line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Nesting Parentheses and Commas

The *actual_arg_list* can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

Token Pasting with

You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers.

Converting to Strings with

The # symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement.

Using the Backslash (\) for Line Continuation

A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

Side Effects and Other Dangers

The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Header File Search with <header_name>

The <*header_name*> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header File Search with "header_name"

The "*header_name*" version specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the include directories as in the <*header_name*> situation.

Global Variables

Borland C++ provides you with predefined global variables for many common needs, such as dates, times, command-line arguments, and so on. For a list of obsolete global variables, see Obsolete Global Variables.

_8087	_osminor
_argc	_osversion
_argv	_psp
_ctype	_sys_errlist
_daylight	_sys_nerr
_directvideo	_threadid
_doserrno	__throwExceptionName
_environ	__throwFileName
_errno	__throwLineNumber
_floatconvert	_timezone
_fmode	_tzname
_new_handler	_version
_osmajor	_wscroll

_8087

Portability Global Variables

Syntax

```
extern int _8087;
```

Header File

dos.h

Description

The *_8087* variable is set to a nonzero value if the startup code autodetection logic detects a floating-point coprocessor.

_8087 Value	Math Coprocessor
1	*8087*
2	80287
3	80387
0	(none detected)

The autodetection logic can be overridden by setting the 87 environment variable to YES or NO. (The commands are `SET 87=YES` and `SET 87=NO`; it is essential that there be no spaces before or after the equal sign.) In this case, the *_8087* variable will reflect the override.

_argc

Portability Example Global Variables

Syntax

```
extern int _argc;
```

Header File

dos.h

Description

_argc has the value of *argc* passed to *main* when the program starts.

/* _argc and _argv example */

```
#include <iostream.h>
#include <dos.h>     // TO GET THE GLOBAL _arg VALUES

void func() {
   cout << "argc= " << _argc << endl;

   for (int i = 0; i < _argc; ++i)
      cout << _argv[i] << endl;
   }

void main(int argc, char ** argv) {
   func(); // THIS FUNCTION KNOWS ALL THE main() ARGUMENTS
   }
```

_argv, _wargv

Portability Example Global Variables

Syntax

```
extern char **_argv;
extern wchar_t ** _wargv
```

Header File

dos.h

Description

_argv points to an array containing the original command-line arguments (the elements of *argv[]*) passed to *main* when the program starts.

_wargv is the Unicode version of *_argv*.

_ctype

Portability Global Variables

Syntax

```
extern char _ctype[];
```

Header File

ctype.h

Description

_ctype is an array of character attribute information indexed by ASCII value + 1. Each entry is a set of bits describing the character. This array is used by isdigit, isprint, and so on.

_daylight

See also Portability Global Variables

Syntax

```
extern int _daylight;
```

Header File

time.h

Description

_daylight is used by the time and date functions. It is set by the tzset, ftime, and localtime functions to 1 for daylight saving time, 0 for standard time.

On Win32, the value of *_daylight* is obtained from the operating system.

_directvideo

Portability Global Variables

Syntax

```
extern int _directvideo;
```

Header File

conio.h

Description

_directvideo controls whether your program's console output goes directly to the video RAM (*_directvideo* = 1) or goes via ROM BIOS calls (*_directvideo* = 0).

The default value is *_directvideo* = 1 (console output goes directly to video RAM). To use *_directvideo* = 1, the video hardware on your system must be identical to IBM display adapters. Setting *_directvideo* = 0 allows your console output to work on any system that is IBM BIOS-compatible.

_directvideo should be used only in character-based applications. It is not allowed in 16-bit Windows, Win32s, or Win32 GUI applications.

_environ, _wenviron

See also Portability Global Variables

Syntax

```
extern char ** _environ;
extern wchar_t ** _wenviron
```

Header File

dos.h

Description

_environ is an array of pointers to strings; it is used to access and alter the operating system environment variables. Each string is of the form:

```
envvar = varvalue
```

where *envvar* is the name of an environment variable (such as PATH), and *varvalue* is the string value to which *envvar* is set (such as C:\BIN;C:\DOS). The string *varvalue* can be empty.

When a program begins execution, the operating system environment settings are passed directly to the program. Note that *env*, the third argument to *main*, is equal to the initial setting of *_environ*.

The *_environ* array can be accessed by getenv; however, the putenv function is the only routine that should be used to add, change or delete the *_environ* array entries. This is because modification can resize and relocate the process environment array, but *_environ* is automatically adjusted so that it always points to the array.

errno

Portability Example Global Variables

Syntax

```
extern int errno;
```

Header File

errno.h

Description

errno is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

When an error in a math or system call occurs, *errno* is set to indicate the type of error. Sometimes *errno* and _doserrno are equivalent. At other times, *errno* does not contain the actual operating system error code, which is contained in *_doserrno*. Still other errors might occur that set only *errno*, not *_doserrno*.

/* errno, _doserrno, _sys_errlist, and _sys_nerr example */

```
/* DISPLAY THE SYSTEM ERRORS. */
#include <errno.h>
#include <stdio.h>

extern char *_sys_errlist[];

main()
{
   int i = 0;

   while(_sys_errlist[i++]) printf("%s\n", _sys_errlist[i]);
   return 0;
}
```

_doserrno

Portability Example Global Variables

Syntax

```
extern int _doserrno;
```

Header File

errno.h

Description

_doserrno is a variable that maps many operating system error codes to errno; however, perror does not use _doserrno directly.

When an operating system call results in an error, *_doserrno* is set to the actual operating system error code. *errno* is a parallel error variable inherited from UNIX.

The following list gives mnemonics for the actual DOS error codes to which *_doserrno* can be set. (This value of *_doserrno* may or may not be mapped (through *errno*) to an equivalent error message string in _sys_errlist.

Mnemonic	DOS error code
E2BIG	Bad environ
EACCES	Access denied
EACCES	Bad access
EACCES	Is current dir
EBADF	Bad handle
EFAULT	Reserved
EINVAL	Bad data
EINVAL	Bad function
EMFILE	Too many open
ENOENT	No such file or directory
ENOEXEC	Bad format
ENOMEM	Mcb destroyed
ENOMEM	Out of memory
ENOMEM	Bad block
EXDEV	Bad drive
EXDEV	Not same device

Refer to your DOS reference manual for more information about DOS error return codes.

_sys_errlist

Portability Example Global Variables

Syntax

```
extern char * _sys_errlist[ ];
```

Header File

errno.h

Description

_sys_errlist is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

To provide more control over message formatting, the array of message strings is provided in *_sys_errlist*. You can use *errno* as an index into the array to find the string corresponding to the error number. The string does not include any newline character.

The following table gives mnemonics and their meanings for the values stored in *_sys_errlist*. The list is alphabetically ordered for ease your reading convenience. For the numerical ordering, see the header file errno.h.

Mnemonic	16-bit Description	32-bit Description
E2BIG	Arg list too long	Arg list too long
EACCES	Permission denied	Permission denied
EBADF	Bad file number	Bad file number
ECHILD		No child process
ECONTR	Memory blocks destroyed	Memory blocks destroyed
ECURDIR	Attempt to remove CurDir	Attempt to remove CurDir
EDEADLOCK		Locking violation
EDOM	Domain error	Math argument
EEXIST	File already exists	File already exists
EFAULT	Unknown error	Unknown error
EINTR		Interrupted function call
EINVACC	Invalid access code	Invalid access code
EINVAL	Invalid argument	Invalid argument
EINVDAT	Invalid data	Invalid data
EINVDRV	Invalid drive specified	Invalid drive specified
EINVENV	Invalid environment	Invalid environment
EINVFMT	Invalid format	Invalid format
EINVFNC	Invalid function number	Invalid function number
EINVMEM	Invalid memory block address	Invalid memory block address
EIO		input/output error
EMFILE	Too many open files	Too many open files
ENAMETOOLONG		File name too long
ENFILE		Too many open files
ENMFILE	No more files	No more files
ENODEV	No such device	No such device
ENOENT	No such file or directory	No such file or directory
ENOEXEC	Exec format error	Exec format error
ENOFILE	No such file or directory	File not found
ENOMEM	Not enough memory	Not enough core
ENOPATH	Path not found	Path not found
ENOSPC		No space left on device
ENOTSAM	Not same device	Not same device
ENXIO		No such device or address
EPERM		Operation not permitted
EPIPE		Broken pipe
ERANGE	Result out of range	Result too large
EROFS		Read-only file system

ESPIPE		Illegal seek
EXDEV	Cross-device link	Cross-device link
EZERO	Error 0	Error 0

Refer to your DOS reference manual for more information about DOS error return codes.

_sys_nerr

Portability Example Global Variables

Syntax

```
extern int _sys_nerr;
```

Header File

errno.h

Description

_sys_nerr is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

This variable is defined as the number of error message strings in _sys_errlist.

_floatconvert

Portability Example Global Variables

Syntax

```
extern int _floatconvert;
```

Header File

stdio.h

Description

Floating-point output requires linking of conversion routines used by *printf*, *scanf*, and any variants of these functions. In order to reduce executable size, the floating-point formats are not automatically linked. However, this linkage is done automatically whenever your program uses a mathematical routine or the address is taken of some floating-point number. If neither of these actions occur, the missing floating-point formats can result in a run-time error.

/* _floatconvert example */

```
/* PREPARE TO OUTPUT FLOATING-POINT NUMBERS. */
#include <stdio.h>
#pragma extref _floatconvert

void main() {
  printf("d = %lf\n", 1);
}
```

_fmode

Portability Global Variables

Syntax

```
extern int _fmode;
```

Header File

fcntl.h

Description

_fmode determines in which mode (text or binary) files will be opened and translated. The value of ***_fmode*** is O_TEXT by default, which specifies that files will be read in text mode. If ***_fmode*** is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.)

In text mode, carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character (LF) on input. On output, the reverse is true: LF characters are translated to CR/LF combinations.

In binary mode, no such translation occurs.

You can override the default mode as set by ***_fmode*** by specifying a *t* (for text mode) or *b* (for binary mode) in the argument *type* in the library functions *fopen, fdopen*, and *freopen*. Also, in the function *open*, the argument access can include either O_BINARY or O_TEXT, which will explicitly define the file being opened (given by the *open pathname* argument) to be in either binary or text mode.

_new_handler

Portability Global Variables

Syntax

```
typedef void (*pvf)();
pvf _new_handler;
```

Header File

new.h

Description

_new_handler contains a pointer to a function that takes no arguments and returns **void**. If **operator new()** is unable to allocate the space required, it will call the function pointed to by *_new_handler*; if that function returns it will try the allocation again. By default, the function pointed to by *_new_handler* simply terminates the application. The application can replace this handler, however, with a function that can try to free up some space. This is done by assigning directly to *_new_handler* or by calling the function set_new_handler,, which returns a pointer to the former handler.

As an alternative, you can set using the function *set_new_handler*, like this:

```
pvf set_new_handler(pvf p);
```

_new_handler is provided primarily for compatibility with C++ version 1.2. In most cases this functionality can be better provided by overloading **operator new()**.

_osmajor

See also Portability Global Variables

Syntax

```
extern unsigned char _osmajor;
```

Header File

dos.h

Description

The major version number of the operating system is available individually through *_osmajor*. For example, if you are running DOS version 3.2, *_osmajor* will be 3.

This variable can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew, ioctl, and _rtl_open.

_osminor

See also Portability Global Variables

Syntax

```
extern unsigned char _osminor;
```

Header File

dos.h

Description

The minor version number of the operating system is available individually through *_osminor*. For example, if you are running DOS version 3.2, *_osminor* will be 20.

This variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew, ioctl, and _rtl_open.

_osversion

See also Portability Global Variables

Syntax

```
extern unsigned _osversion;
```

Header File

dos.h

Description

_osversion contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the *x* is the major version number, and *y* is the minor version number.)

_osversion is functionally identical to _version.

_psp

Portability Global Variables

Syntax

```
extern unsigned int _psp;
```

Header Files

dos.h

process.h

stdlib.h

Description

_psp specifies the address of the program segment prefix (PSP) of a program. The PSP is a DOS process descriptor; it contains initial DOS information about the program.

Note: *_psp* cannot be used in DLLs.

_ _throwExceptionName

See also Portability Global Variables

Syntax

```
extern char * _ _throwExceptionName;
```

Header File

except.h

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

_ _throwFileName

See also Portability Global Variables

Syntax

```
extern char * _ _throwFileName;
```

Header File

except.h

Description

Use this global variable to get the name of a thrown exception. The output for this variables is a printable character string.

To get the file name for a thrown exception with *__throwFileName*, you must compile the module with the `-xp` compiler option.

_ _throwLineNumber

See also Portability Global Variables

Syntax

```
extern char * _ _throwLineNumber;
```

Header File

except.h

Description

Use this global variable to get the name of a thrown exception. The output for this variables is a printable character string.

To get the line number for a thrown exception with *__throwLineNumber*, you must compile the module with the `-xp` compiler option.

_threadid

Portability Global Variables

Syntax

```
extern long _threadid;
```

Header File

stddef.h

Description

_threadid is a long integer that contains the ID of the currently executing thread. It is implemented as a macro, and should be declared only by including stddef.h.

_timezone

See also Portability Global Variables

Syntax

```
extern long _timezone;
```

Header File

time.h

Description

_timezone is used by the time-and-date functions. It is calculated by the tzset function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich mean time.

On Win32, the value of ***_timezone*** is obtained from the operating system.

_tzname, _wtzname

See also Portability Global Variables

Syntax

```
extern char * _tzname[2]
extern wchar_t *const _wtzname[2]
```

Header File

time.h

Description

The global variable *_tzname* is an array of pointers to strings containing abbreviations for time zone names. *_tzname*[0] points to a three-character string with the value of the time zone name from the *TZ* environment string. The global variable *_tzname*[1] points to a three-character string with the value of the daylight saving time zone name from the *TZ* environment string. If no daylight saving name is present, *_tzname*[1] points to a null string.

On Win32, the value of *_tzname* is obtained from the operating system.

_version

See also Portability Global Variables

Syntax

```
extern unsigned _version;
```

Header File

dos.h

Description

_version contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version *x.y*, the *x* is the major version number, and *y* is the minor.)

_wscroll

Portability Global Variables

Syntax

```
extern int _wscroll
```

Header File

conio.h

Description

_wscroll is a console I/O flag. Its default value is 1. If you set *_wscroll* to 0, scrolling is disabled. This can be useful for drawing along the edges of a window without having your screen scroll.

_wscroll should be used only in character-based applications. It is available for EasyWin but is not allowed in 16-bit Windows, Win32s, or Win32 GUI applications.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

Obsolete Global Variables

See also

The following global variables have been renamed to comply with ANSI naming requirements. You should always use the new names.

If you link with libraries that were compiled with Borland C++ 3.1 (or earlier) header files, the following message is generated:

```
Error: undefined external <varname> in module <LIBNAME>.LIB
```

You should recompile a library module that results in such an error. If you cannot recompile the code for such libraries, you can link with `OBSOLETE.LIB` to resolve the external variable names.

The following global variables have been renamed:

Old Name	New Name	Header File
daylight	_daylight	time.h
directvideo	_directvideo	conio.h
environ	_environ	stdlib.h
sys_errlist	_sys_errlist	errno.h
sys_nerr	_sys_nerr	errno.h
timezone	_timezone	time.h
tzname	_tzname	time.h

_mexcep (typedef)

See also

Defined In
math.h

Description
The typedef *_mexcep* enumerates these constants that represent possible mathematical errors.

_mexcep Constant	Mathematical error
DOMAIN	Argument was not in domain of function
	Example: `log(-1)`
SING	Argument would result in a singularity
	Example: `pow(0, -2)`
OVERFLOW	Argument would produce a function result > MAXDOUBLE
	Example: `exp(1000)`
UNDERFLOW	Argument would produce a function result < MINDOUBLE
	Example: `exp(-1000)`
TLOSS	Argument would produce function result with total loss of significant digits
	Example: `sin(10**70)`

The symbolic constants MAXDOUBLE and MINDOUBLE are defined in VALUES.H.

div_t and ldiv_t (typedef struct)

Defined In
stdlib.h

Syntax
```
typedef struct{
  int quot;     /* quotient */
  int rem;      /* remainder */
} div_t;
```

Description
The *div_t* type is a structure of integers used by div.

Syntax
```
typedef struct{
  long int quot;     /* quotient */
  long int rem;      /* remainder */
} ldiv_t;
```

Description
The *ldiv_t* type is a structure of **long**s used by ldiv.

jmp_buf (typedef struct)

Defined In
setjmp.h

Syntax
```
typedef struct{
  unsigned  j_sp,   j_ss;
  unsigned  j_flag, j_cs;
  unsigned  j_ip,   j_bp;
  unsigned  j_di,   j_es;
  unsigned  j_si,   j_ds;
} jmp_buf[1];
```

Description
A buffer of type *jmp_buf* is used to save and restore the program task state.

FILE (typedef struct)

Defined In
stdio.h

Syntax
```
typedef struct{
  short          level;
  unsigned       flags;
  char           fd;
  unsigned char  hold;
  short          bsize;
  unsigned char *buffer, *curp;
  unsigned       istemp;
  short          token;
} FILE;
```

Description
File control structure for streams.

dosSearchInfo (typedef struct)

Defined In
dos.h

Declaration

```
typedef struct{
  char   drive;
  char   pattern [13];
  char   reserved [7];
  char   attrib;
  short  time;
  short  date;
  long   size;
  char   nameZ [13];
} dosSearchInfo;
```

atexit_t (type)

Defined In
stdlib.h

Syntax
```
typedef void (* atexit_t)(void);
```

Description
Type of exit function passed to atexit.

size_t (type)

Defined in
stddef.h

Description
Type which is an unsigned integer returned by the **sizeof** operator.

ptrdiff_t (type)

Defined In
stddef.h

Description
A signed integer that represents the result of subtracting two pointers.

fpos_t (type)

Defined In
stdio.h

Description
A file position type.

time_t (type)

Defined In
time.h
sys\types.h

Description
This variable type defines the value used by the time functions declared in time.h.

The *time_t* value will not work after the hour of 3:14:07 on the year 01/19/2038.

The functions that use the *time_t* value are as follows:

```
char ctime(const time_t  *time);
double difftime(time_t time2, time_t time1);
struct tm * gmtime(const time_t  *timer);
struct tm * localtime(const time_t  *timer);
time_t time(time_t  *timer);
time_t mktime(struct tm  *timeptr);
int    stime(time_t  *tp);
```

`clock_t (type)`

Defined In
time.h

Description
The CLK_TCK constant defines the number of clock ticks per second. This data type is returned by the clock function stores an elapsed time measured in clock ticks.

sig_atomic_t (type)

Defined In
signal.h

Description
Atomic entity type.

wchar_t (type)

Defined In
stddef.h

Description
Wide-character constant (C only). In a C++ file, **wchar_t** is a keyword.

A character constant preceded by an *L* is a wide-character constant.

O_xxxx #defines

Header File
fcntl.h

Description
These #defines are bit definitions for a file-access argument.

These RTL file-open functions use some (not all) of these definitions:

- _dos_open
- fdopen
- fopen
- freopen
- _fsopen
- open
- _rtl_open
- sopen

_dos_open and sopen also use file-sharing symbolic constants in the file-access argument.

Category	Constant	Description
Read/Write flag (Used by _dos_open, open, _rtl_open, and sopen)		
	O_RDONLY	Open for reading only
	O_WRONLY	Open for writing only
	O_RDWR	Open for reading and writing
Other access flags (Used by open and sopen)		
	O_NDELAY	Not used; for UNIX compatibility.
	O_APPEND	Append to end of file If set, the file pointer is set to the end of the file prior to each write.
	O_CREAT	Create and open file If the file already exists, has no effect. If the file does not exist, the file is created.
	O_EXCL	Exclusive open: Used only with O_CREAT. If the file already exists, an error is returned.
	O_TRUNC	Open with truncation If the file already exists, its length is truncated to 0. The file attributes remain unchanged.
Binary-mode/Text-mode flags(Used by fdopen, fopen, freopen, _fsopen, open and sopen)		
	O_BINARY	No translation: Explicitly opens the file in binary mode
	O_TEXT	CR-LF translation: Explicitly opens the file in text mode
Additional values available under DOS 3.x (Used by _rtl_open)		
	O_NOINHERIT	Child processes inherit file
	O_DENYALL	Error if opened for read/write
	O_DENYWRITE	Error if opened for write
	O_DENYREAD	Error if opened for read
	O_DENYNONE	Allow concurrent access

Note: Only one of the O_DENYxxx options can be included in a single open. These file-sharing attributes are in addition to any locking performed on the files.

DO NOT MODIFY these special read-only bits described in DOS documentation!

O_CHANGED	Special DOS read-only bit
O_DEVICE	Special DOS read-only bit

SEEK_xxx

See also

Header File

io.h

stdio.h

Description

#defines that set seek starting points

Constant	Value	File Location
SEEK_SET	0	Seeks from beginning of file
SEEK_CUR	1	Seeks from current position
SEEK_END	2	Seeks from end of file

SH_xxxx

Header File
share.h

Description
File-sharing mode for use with _dos_open and sopen (under DOS 3.0 or later).

Constant	Meaning
SH_COMPAT	Sets compatibility mode:
	Allows other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode.
SH_DENYNONE	Permits read/write access
	Allows other shared opens to the file, but not other SH_COMPAT opens
SH_DENYNO	Permits read/write access (provided for compatibility)
SH_DENYRD	Denies read access. Allows only writes from any other open to the file
SH_DENYRW	Denies read/write access. Only the current handle may have access to the file
SH_DENYWR	Denies write access. Allows only reads from any other open to the file
O_NOINHERIT	The file is not passed to child programs

These file-sharing attributes are in addition to any locking performed on the files.

P_xxxx

Header File
process.h

Description
Modes used by the spawn... functions.

Constant	Meaning
P_WAIT	Child runs separately, parent waits until exit
P_DETACH	Child and parent run concurrently with child process in background mode
P_NOWAIT	Child and parent run concurrently (Not implemented)
P_NOWAITO	Child and parent run concurrently, but the child process is not saved
P_OVERLAY	Child replaces parent so that parent no longer exists

SIG_xxx

See also

Header File
signal.h

Description
Predefined functions for handling signals generated by raise or by external events.

Name	Meaning
SIG_DFL	Terminate the program
SIG_IGN	No action, ignore signal
SIG_ERR	Return error code

SIGxxxx

Header File
signal.h

Description
Signal types used by raise and signal.

Signal	Note	Meaning	Default Action
SIGABRT	(*)	Abnormal termination	= to calling `_exit(3)`
SIGFPE		Bad floating-point operation Arithmetic error caused by division by 0, invalid operation, etc.	= to calling `_exit(1)`
SIGILL	(#)	Illegal operation	= to calling `_exit(1)`
SIGINT		Control-C interrupt	Is to do an INT 23h
SIGSEGV	(#)	Invalid access to storage	= to calling `_exit(1)`
SIGTERM	(*)	Request for program termination	= to calling _exit(1)

(*) Signal types marked with a (*) aren't generated by DOS or Borland C++ during normal operation. However, they can be generated with raise.

(#) Signals marked by (#) can't be generated asynchronously on 8088 or 8086 processors but can be generated on some other processors (see signal for details).

stdaux, stderr, stdin, stdout, and stdprn

Header File
stdio.h

Description
Predefined streams automatically opened when the program is started.

Name	Meaning
stdin	Standard input device
stdout	Standard output device
stderr	Standard error output device
stdaux	Standard auxiliary device
stdprn	Standard printer

S_Ixxxx

Header File
sys\stat.h

Description
Definitions used for file status and directory functions.

Name	Meaning
S_IFMT	File type mask
S_IFDIR	Directory
S_IFIFO	FIFO special
S_IFCHR	Character special
S_IFBLK	Block special
S_IFREG	Regular file
S_IREAD	Owner can read
S_IWRITE	Owner can write
S_IEXEC	Owner can execute

NULL #define

Header File
stddef.h

Description
Null pointer constant that is compatible with any data object pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any data object defined within the program.

Bit Definitions for fnsplit

Header File
dir.h

Description
Bit definitions returned from fnsplit to identify which pieces of a file name were found during the split.

Flag	Component
DIRECTORY	Path includes a directory (and possibly subdirectories)
DRIVE	Path includes a drive specification (see DIR.H)
EXTENSION	Path includes an extension
FILENAME	Path includes a file name
WILDCARDS	Path contains wildcards (`*` or `?`)

MAXxxxx

Header File
dir.h

Description
These symbols define the maximum number of characters in a file specification for fnsplit (including room for a terminating NULL).

Name	Meaning
MAXPATH	Complete file name with path
MAXDRIVE	Disk drive (e.g., "A:")
MAXDIR	File subdirectory specification
MAXFILE	File name without extension
MAXEXT	File extension

_F_xxxx

Header File
stdio.h

Description
File status flags of streams

Name	Meaning
_F_RDWR	Read and write
_F_READ	Read-only file
_F_WRIT	Write-only file
_F_BUF	Malloc'ed buffer data
_F_LBUF	Line-buffered file
_F_ERR	Error indicator
_F_EOF	EOF indicator
_F_BIN	Binary file indicator
_F_IN	Data is incoming
_F_OUT	Data is outgoing
_F_TERM	File is a terminal

FA_xxxx

Header File
dos.h

Description
DOS file attributes

Constant	Description
FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

For more detailed information about these attributes, refer to your DOS reference manuals.

EXIT_xxxx

Header File
stdlib.h

Description
Constants defining exit conditions for calls to the exit function.

Name	Meaning
EXIT_SUCCESS	Normal program termination
EXIT_FAILURE	Abnormal program termination

_IOxxx

See also

Header File
stdio.h

Description
Constants for defining buffering style to be used with a file.

Name	Meaning
_IOFBF	The file is fully buffered. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.
_IOLBF	The file is line buffered. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	The file is unbuffered. The buf and size parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

BUFSIZ

Header File
stdio.h

Description
Default buffer size used by setbuf function.

EOF

Header File
stdio.h

Description
A constant indicating that end-of-file has been reached on a file.

_IS_xxx

Header File
ctype.h

Description
Bit settings in the _ctype[] used by the is... character macros.

Name	Meaning
_IS_SP	Is space
_IS_DIG	Is digit
_IS_UPP	Is uppercase
_IS_LOW	Is lowercase
_IS_HEX	[A-F] or [a-f]
_IS_CTL	Control
_IS_PUN	Punctuation

CHAR_xxx

Header File
LIMITS.H

Description

Name	Meaning
CHAR_BIT	Type char, number of bits
CHAR_MAX	Type char, minimum value
CHAR_MIN	Type char, maximum value

These values are independent of whether type char is defined as signed or unsigned by default.

SCHAR_xxx

Header File
limits.h

Description

Name	Meaning
SCHAR_MAX	Type char, maximum value
SCHAR_MIN	Type char, minimum value

Uxxxx_MAX

Header File
limits.h

Description

Name	Maximum value for type xxx
UCHAR_MAX	unsigned char
USHRT_MAX	unsigned short
UINT_MAX	unsigned integer
ULONG_MAX	unsigned long

SHRT_xxx

Header File
limits.h

Description

Name	Meaning
SHRT_MAX	Type short, maximum value
SHRT_MIN	Type short, minimum value

INT_xxx

Header File
limits.h

Description
Maximum and minimum value for type int.

Name	Meaning
INT_MAX	Type int, maximum value
INT_MIN	Type int, minimum value

LONG_xxx

Header File
limits.h

Description
Maximum and minimum value for type long.

Name	Meaning
LONG_MAX	Type long, maximum value
LONG_MIN	Type long, minimum value

CW_DEFAULT

Header File
float.h

Description
Default control word for 8087/80287 math coprocessor.

EDOM, ERANGE, HUGE_VAL

Header File

errno.h

math.h

Description

Name	Meaning
EDOM	Error code for math domain error
ERANGE	Error code for result out of range
HUGE_VAL	Overflow value for math functions

NDEBUG

Header File
assert.h

Description
NDEBUG means "Use #define to treat assert as a macro or a true function".

Can be defined in a user program. If defined, assert is a true function; otherwise assert is a macro.

NFDS

Header File
dos.h

Description
Maximum number of file descriptors.

MAXxxxx

Header File
values.h

Description
Maximum values for integer data types

Name	Meaning
MAXSHORT	Largest short
MAXINT	Largest int
MAXLONG	Largest long

M_E, M_LOGxxx, M_LNxx

Header File
math.h

Description
The constant values for logarithm functions.

Name	Meaning
M_E	The value of e
M_LOG2E	The value of log(e)
M_LOG10E	The value of log10(e)
M_LN2	The value of ln(2)
M_LN10	The value of ln(10)

PI constants

Header File
math.h

Description
Common constants of pi

Name	Meaning
M_PI	pi
M_PI_2	One-half pi
M_PI_4	One-fourth pi
M_1_PI	One divided by pi
M_2_PI	Two divided by pi
M_1_SQRTPI	One divided by the square root of pi
M_2_SQRTPI	Two divided by the square root of pi

M_SQRTxxx

Header File
math.h

Description
Constant values for square roots of 2.

Name	Meaning
M_SQRT2	Square root of 2
M_SQRT_2	1/2 the square root of 2

L_ctermid

Header File
stdio.h

Description
The length of a device id string.

L_tmpnam

Header File
stdio.h

Description
Size of an array large enough to hold a temporary file name string.

TMP_MAX

Header File
stdio.h

Description
Maximum number of unique file names.

OPEN

Header File
stdio.h

Description
Number of files that can be open simultaneously.

Name	Meaning
FOPEN_MAX	Maximum files per process
SYS_OPEN	Maximum files for system

HANDLE_MAX

Header File
io.h

Description
Maximum number of handles.

RAND_MAX

Header File
stdlib.h

Syntax

Description
Maximum value returned by rand function.

BITSPERBYTE

Header File
values.h

Description
Number of bits in a byte.

Float and Double Limits

Header File
values.h

Description

UNIX System V compatible:

_LENBASE	Base to which exponent applies

Limits for double float values

_DEXPLEN	Number of bits in exponent
DMAXEXP	Maximum exponent allowed
DMAXPOWTWO	Largest power of two allowed
DMINEXP	Minimum exponent allowed
DSIGNIF	Number of significant bits
MAXDOUBLE	Largest magnitude double value
MINDOUBLE	Smallest magnitude double value

Limits for float values

_FEXPLEN	Number of bits in exponent
FMAXEXP	Maximum exponent allowed
FMAXPOWTWO	Largest power of two allowed
FMINEXP	Minimum exponent allowed
FSIGNIF	Number of significant bits
MAXFLOAT	Largest magnitude float value
MINFLOAT	Smallest magnitude float value

HIBITxxx

Header File
values.h

Description
Bit mask for the high (sign) bit of standard integer types.

Name	Meaning
HIBITS	For type short
HIBITI	For type int
HIBITL	For type long

Error Numbers in errno

Header File
errno.h

Description
These are the mnemonics and meanings for the error numbers found in errno.

Each value listed can be used to index into the sys_errlist array for displaying messages.

Also, perror will display messages.

Mnemonic	Meaning
EZERO	Error 0
EINVFNC	Invalid function number
ENOFILE	File not found
ENOPATH	Path not found
ECONTR	Memory blocks destroyed
EINVMEM	Invalid memory block address
EINVENV	Invalid environment
EINVFMT	Invalid format
EINVACC	Invalid access code
EINVDAT	Invalid data
EINVDRV	Invalid drive specified
ECURDIR	Attempt to remove CurDir
ENOTSAM	Not same device
ENMFILE	No more files
ENOENT	No such file or directory
EMFILE	Too many open files
EACCES	Permission denied
EBADF	Bad file number
ENOMEM	Not enough memory
ENODEV	No such device
EINVAL	Invalid argument
E2BIG	Arg list too long
ENOEXEC	Exec format error
EXDEV	Cross-device link
EDOM	Math argument
ERANGE	Result too large
EFAULT	Unknown error
EEXIST	File already exists

Bit fields

A bit field is an element of a structure that is defined in terms of bits. Using a special type of struct definition, you can declare a structure element that can range from 1 to 16 bits in length.

For example, this struct

```
struct  bit_field {
  int bit_1        : 1;
  int bits_2_to_5  : 4;
  int bit_6        : 1;
  int bits_7_to_16 : 10;
} bit_var;
```

corresponds to this collection of bit fields:

```
|---------------------------|---|---------|---|
| 16 15 14 13 12 11 10 9 8 7 | 6 | 5 4 3 2 | 1 |
|---------------------------|---|---------|---|
```

See Also

class

union

ftime

header: IO.H

A file's time and date. Used by the functions getftime and setftime.

```
struct ftime {
  unsigned ft_tsec  : 5;  /* Two seconds */
  unsigned ft_min   : 6;  /* Minutes */
  unsigned ft_hour  : 5;  /* Hours */
  unsigned ft_day   : 5;  /* Days */
  unsigned ft_month : 4;  /* Months */
  unsigned ft_year  : 7;  /* Year - 1980 */
};
```

_exception and _exceptionl
header: MATH.H

The format of error information for math routines.

Struct exception is used by _matherr:

```
struct _exception {
  int     type;
  char   *name;
  double  arg1, arg2, retval;
};
```

Struct _exceptionl is used by _matherrl:

```
struct _exceptionl {
  int     type;
  char   *name;
  long double arg1, arg2, retval;
};
```

Member	What It Is (Or Represents)
type	The type of mathematical error that occurred; an enum type defined in the typedef _mexcep.
name	A pointer to a null-terminated string holding the name of the math library function that resulted in an error.
arg1,	The arguments (passed to the function *name) that caused the error.
arg2	If only one argument was passed to the function, it is stored in arg1.
retval	The default return value for matherr; you can modify this value.

complex and _complexl

header: MATH.H

Complex number representation.

Struct complex is used by the complex function cabs.

```
struct complex {
  double  x, y;
};
```

Struct _complexl is used by the long double complex function cabsl.

```
struct _complexl {
  long double x, y;
};
```

x is the real part, and y is the imaginary part.

tm

header: TIME.H

A structure defining the broken-down time.

Used by the functions asctime, gmtime,localtime, mktime, and strftime.

```
struct tm {
  int tm_sec;   /* Seconds */
  int tm_min;   /* Minutes */
  int tm_hour;  /* Hour (0--23) */
  int tm_mday;  /* Day of month (1--31) */
  int tm_mon;   /* Month (0--11) */
  int tm_year;  /* Year (calendar year minus 1900) */
  int tm_wday;  /* Weekday (0--6; Sunday = 0) */
  int tm_yday;  /* Day of year (0--365) */
  int tm_isdst; /* 0 if daylight savings time is not in effect) */
};
```

timeb

header: SYS\TIMEB.H

Current time information filled out by the ftime function.

```
struct timeb {
  long time ;      /* seconds since 00:00:00, 1/1/70, GMT */
  short millitm ;  /* fraction of second  (in milliseconds) */
  short timezone ; /* difference between local time and GMT */
  short dstflag ;  /* 0 if daylight savings time is not in effect */
};
```

timezone is computed going west from GMT.

ftime gets this field from timezone, which is set by tzset.

stat

header: SYS\STAT.H

A structure containing information about a file or directory. Used by the fstat and stat functions.

```
struct stat {
  short  st_dev,   st_ino;
  short  st_mode,  st_nlink;
  int    st_uid,   st_gid;
  short  st_rdev;
  long   st_size,  st_atime;
  long   st_mtime, st_ctime;
};
```

Element	What It Is
st_dev	Drive number of disk containing the file, or file handle if the file is on a device
st_mode	Bit mask giving information about the open file's mode
st_nlink	Set to the integer constant 1
st_rdev	Same as st_dev
st_size	Size of the open file in bytes
st_atime	Most recent time the open file was modified
st_mtime	Same as st_atime
st_ctime	Same as st_atime
st_ino	These elements contain values
st_uid	that are not meaningful under
st_gid	DOS.

ffblk

header: DIR.H

DOS file control block structure.

```
struct ffblk {
  char ff_reserved[21]; /* reserved by DOS */
  char ff_attrib;       /* attribute found */
  int  ff_ftime;        /* file time */
  int  ff_fdate;        /* file date */
  long ff_fsize;        /* file size */
  char ff_name[13];     /* found file name */
};
```

Remarks

ff_ftime and ff_fdate are 16-bit structures divided into bit fields for referring to the current date and time.

The structure of these fields was established by DOS.

See Also

findfirst

ftime structure

find_t structure

fcb

header: DOS.H

The structure of the MS-DOS file control blocks.

```
struct fcb {
  char   fcb_drive;
  char   fcb_name[8],  fcb_ext[3];
  short  fcb_curblk,   fcb_recsize;
  long   fcb_filsize;
  short  fcb_date;
  char   fcb_resv[10], fcb_currec;
  long   fcb_random;
};
```

xfcb

header: DOS.H

The MS-DOS extended file control block structure.

```
struct xfcb {
  char          xfcb_flag;
  char          xfcb_resv[5];
  char          xfcb_attr;
  struct  fcb   xfcb_fcb;
};
```

dfree

header: DOS.H

The structure of the information returned by the getdfree function.

```
struct dfree {
  unsigned df_avail;  /* Available clusters */
  unsigned df_total;  /* Total clusters */
  unsigned df_bsec;   /* Bytes per sector */
  unsigned df_sclus;  /* Sectors per cluster */
};
```

fatinfo

header: DOS.H

The structure of the file allocation table information filled in by the getfat and getfatd functions.

```
struct fatinfo {
  char  fi_sclus;  /* sectors per cluster */
  char  fi_fatid;  /* the FAT id byte */
  int   fi_nclus;  /* number of clusters */
  int   fi_bysec;  /* bytes per sector */
};
```

time

header: DOS.H

Structure of the time as used by these functions:

dostounix

gettime

settime

unixtodos

```
struct time {
  unsigned char  ti_min;   /* minutes */
  unsigned char  ti_hour;  /* hours */
  unsigned char  ti_hund;  /* hundredths of seconds */
  unsigned char  ti_sec;   /* seconds */
};
```

date

header: DOS.H

Structure of the date as used by these functions:

dostounix

getdate

setdate

unixtodos

```
struct date {
  int da_year;     /* current year */
  char da_day;     /* day of the month */
  char da_mon;     /* month (1 = Jan) */
};
```

REGS (union)
header: DOS.H

The union REGS is used to pass information to and from these functions:

int86

int86x

intdos

intdosx

```
union REGS {
  struct  WORDREGS  x;
  struct  BYTEREGS  h;
};
```

See Also
struct REGPACK

BYTEREGS and WORDREGS

header: DOS.H

Structures for storing byte and word registers

```
struct BYTEREGS  {
  unsigned char  al, ah, bl, bh;
  unsigned char  cl, ch, dl, dh;
};
struct WORDREGS  {
  unsigned int  ax, bx, cx, dx;
  unsigned int  si, di, cflag, flags;
};
```

SREGS

header: DOS.H

The structure of the segment registers passed to and filled in by these functions:

int86x

intdosx

segread

```
struct SREGS {
  unsigned int  es;
  unsigned int  cs;
  unsigned int  ss;
  unsigned int  ds;
};
```

REGPACK
header: DOS.H

The structure of the values passsed to and returned by the intr function call.

```
struct REGPACK {
  unsigned  r_ax, r_bx, r_cx, r_dx;
  unsigned  r_bp, r_si, r_di;
  unsigned  r_ds, r_es, r_flags;
};
```

See Also
REGS

COUNTRY
header: DOS.H

The structure COUNTRY specifies how certain country-dependent data is to be formatted.

```
struct COUNTRY {
  int co_date;          /* date format */
  char co_curr[5];      /* currency symbol */
  char co_thsep[2];     /* thousands separator */
  char co_desep[2];     /* decimal separator */
  char co_dtsep[2];     /* date separator */
  char co_tmsep[2];     /* time separator */
  char co_currstyle;    /* currency style */
  char co_digits;       /* significant digits in currency */
  char co_time;         /* time format */
  long co_case;         /* case map */
  char co_dasep[2];     /* data separator */
  char co_fill[10];     /* filler */
};
```

This is the date format in co_date:

Value	Style	Format
0	U.S.	(Month, Day, Year)
1	European	(Day, Month, Year)
2	Japanese	(Year, Month, Day)

This is the currency display style in co_currstyle:

Style	Example	Meaning
0	`$10.52`	Currency symbol precedes value with no spaces between the symbol and the number
1	`10.52$`	Currency symbol follows value with no spaces between the number and the symbol
2	`$ 10.52`	Currency symbol precedes value with a space after the symbol.
3	`10.52 $`	Currency symbol follows the number with a space before the symbol.

devhdr

header: DOS.H

Header structure for MS-DOS device drivers.

```
struct devhdr {
  long            dh_next;
  short           dh_attr;
  unsigned short  dh_strat;
  unsigned short  dh_inter;
  char            dh_name[8];
};
```

lconv

Used by localeconv.

```
struct lconv {
  char *decimal_point;
  char *thousands_sep;
  char *grouping;
  char *int_curr_symbol;
  char *currency_symbol;
  char *mon_decimal_point;
  char *mon_thousands_sep;
  char *mon_grouping;
  char *positive_sign;
  char *negative_sign;
  char int_frac_digits;
  char frac_digits;
  char p_cs_precedes;
  char p_sep_by_space;
  char n_cs_precedes;
  char n_sep_by_space;
  char p_sign_posn;
  char n_sign_posn;
};
```

DOSERROR
header: DOS.H

Used by dosexterr to return extended DOS errors.

```
struct DOSERROR {
  int  de_exterror;     /* extended error */
  char de_class;        /* error class */
  char de_action;       /* action */
  char de_locus;        /* error locus */
};
```

See Also
_doserrno

errno

Example
DOSERROR

/* DOSERROR example */

```
#include <stdio.h>
#include <dos.h>
  int main(void)  {
  FILE *fp;
  struct DOSERROR info;
  fp = fopen("perror.dat","r");
  if (!fp) perror("Unable to open file for reading");
    dosexterr(&info);
  printf("Extended DOS error information:\n");
  printf("   Extended error:     %d\n",info.de_exterror);
  printf("            Class:     %x\n",info.de_class);
  printf("           Action:     %x\n",info.de_action);
  printf("      Error Locus:     %x\n",info.de_locus);
  return 0;
}
```

find_t

header: DOS.H

DOS file control block structure used by _dos_findfirst and _dos_findnext.

The find_t structure corresponds exactly to the ffblk structure.

```
struct find_t {
  char     reserved[21];   /* Microsoft reserved - do not change*/
  char     attrib;         /* attribute byte for matched file */
  unsigned wr_time;        /* time of last write to file */
  unsigned wr_date;        /* date of last write to file */
  long     size;           /* size of file */
  char     name[13];       /* asciiz name of matched file */
};
```

dirent

header: DIRENT.H

Structure that corresponds to a single directory entry. Used by readdir.

In addition to non-accessible members, dirent contains the member

```
char  d_name[];
```

where d_name is an array of characters containing the null-terminated file name for the current directory entry.

The size of the array is indeterminate; use strlen to determine the length of the file name.

dosdate_t
header: DOS.H

Structure used by _dos_getdate and _dos_setdate.

```
struct dosdate_t {
  unsigned char day;        /* 1--31 */
  unsigned char month;      /* 1--12 */
  unsigned int  year;       /* 1980--2099 */
  unsigned char dayofweek;  /* 0--6; 0 = Sunday */
};
```

dostime_t
header: DOS.H

Structure used by _dos_gettime and _dos_settime.

```
struct dostime_t {
  unsigned char hour;         /* Hours */
  unsigned char minute;       /* Minutes */
  unsigned char second;       /* Seconds */
  unsigned char hsecond;      /* Hundredths of seconds */
};
```

diskfree_t
header: DOS.H

Structure used by _dos_getdiskfree.

```
struct diskfree_t {
  unsigned total_clusters;
  unsigned avail_clusters;
  unsigned sectors_per_cluster;
  unsigned bytes_per_sector;
};
```

utimbuf

header: UTIME.H

Structure used by utime.

```
struct utimbuf {
  time_t   actime;     /* access time */
  time_t   modtime;    /* modification time */
};
```

Because the DOS file system supports only a modification time, utime ignores actime and uses only modtime to set the file's modification time.

Keywords

{button A,JI(`',`keywords_a')} {button B,JI(`',`keywords_b')} {button C,JI(`',`keywords_c')} {button D,JI(`',`keywords_d')} {button E,JI(`',`keywords_e')} {button F,JI(`',`keywords_f')} {button G,JI(`',`keywords_g')} {button H,JI(`',`keywords_h')} {button I,JI(`',`keywords_i')} {button J,JI(`',`keywords_i')} {button I,JI(`',`keywords_l')} {button N,JI(`',`keywords_n')} {button O,JI(`',`keywords_o')} {button P,JI(`',`keywords_p')} {button R,JI(`',`keywords_r')} {button S,JI(`',`keywords_s')} {button T,JI(`',`keywords_t')} {button U,JI(`',`keywords_u')} {button V,JI(`',`keywords_v')} {button W,JI(`',`keywords_w')}

Keywords are words reserved for special purposes and must **not** be used as normal identifier names. You can set options in the IDE or for the command-line compiler to select ANSI keywords only, UNIX keywords only, or to support all keywords--including the Borland C++ extensions.

This is an alphabetical listing of the keywords supported in this release of Borland C++. For a functional listing of the keywords, see Keywords (by Category).

A

__asm
_asm
asm
auto

B

break
bool

C

case
catch
__cdecl
_cdecl
cdecl
char
class
const
const_cast
continue
__cs
_cs

D

__declspec
default
delete
do
double
_ds
__ds
dynamic_cast

E

else
enum
_es
__es
__except
explicit
_export
__export
extern

F

false
__far
_far
far

__fastcall
_fastcall
__finally
float
for
friend

G

goto

H

__huge
_huge
huge

I

if
__import
_import
inline
interrupt
int
__int8
__int16
__int32
__int64

__interrupt
_interrupt

L

__loadds
_loadds
long
mutable

N

namespace
__near
_near
near
new

O

operator

P

__pascal
_pascal
pascal
private
protected
public

R

register
reinterpret_cast
return
__rtti

S

__saveregs

_saveregs
__seg
_seg
short
signed
sizeof
__ss
_ss
static
static_cast
__stdcall
_stdcall
struct
switch

T

template
this
__thread
throw
true
__try
try
typedef
typename
typeid

U

union
using
unsigned

V

virtual
void
volatile

W

wchar_t
while

Keywords (by Category)

This is a categorical listing of the keywords Borland C++ supports. For an alphabetical listing of the keywords, see Keywords (Alphabetical).

Borland C++ Extensions	keywords unique to Borland C++
C++ Specific	keywords recognized only in C++ programs
Modifiers	keywords that change one or more attributes of an identifier associated with an object
Operators	keywords that invoke functions against objects or identifiers
Statements	keywords that specify program control during execution
Storage Class Specifiers	keywords that define the location and duration of an identifier
Type Specifiers	keywords that determine how memory is allocated and bit patterns are interpreted

Borland C++ Keyword Extensions

Borland C++ provides additional keywords that are not part of the ANSI or UNIX conventions. You cannot use these keywords in your programs if you set the IDE or command-line options to recognize only ANSI or UNIX keywords.

The Borland C++ keyword extensions are:

_asm
__asm
_cdecl
cdecl
_cs
__declspec
__ds
_ds
__es
_es
__except
__export
_export
far
_far
__far
__fastcall
_fastcall
__finally
__huge
_huge
huge
_import
__import
__interrupt
_interrupt
interrupt
_loadds
__loadds
near
__near
_near
_pascal
__pascal
pascal
__rtti
__saveregs
_saveregs
__seg
_seg
_ss
__thread
__try

Table of C++ Specific Keywords

There are several keywords specific to C++. They are not available if you are writing a C-only program.

The keywords specific to C++ are:

asm	mutable	this	
bool	namespace	throw	
catch	new	true	
class	operator	try	
const_cast	private	typeid	
delete	explicit	protected	reinterpret_cast
dynamic_cast	public	using	
false	__rtti	virtual	
friend	static_cast	wchar_t	
inline	template	typename	

Modifiers

A declaration uses modifiers to alter aspects of the identifier/object mapping.

The Borland C++ modifiers are:

__cdecl
const
__cs
__declspec
__ds
__es
__export
__far
__fastcall
__huge
__import
__interrupt
__loadds
__near
__pascal
__rtti
__ss
__stdcall
volatile

Operator Keywords

See also

Several Borland C++ keywords denote operators that invoke functions against objects and identifiers.

The keyword operators supported by Borland C++ are:

delete	operator	typeid
new	sizeof	

Statement Keywords

Statements specify the flow of control in a program. In the absence of specific jumps and selection statements, statements execute sequentially as they appear in the source code.

The statement keywords in Borland C++ are:

break	else	switch
case	__finally	throw
catch	for __try	
continue	goto	try
default	if while	
do	return	
__except		

Storage Class Specifiers

Storage classes specifiers are also called *type specifiers*. They dictate the location (data segment, register, heap, or stack) of an object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the declaration syntax, by its placement in the source code, or by both of these factors.

The keyword **mutable** does not affect the lifetime of the class member to which it is applied.

The storage class specifiers in Borland C++ are:

auto	mutable	static
__declspec		
extern	register	typedef

Type Specifiers

The type determines how much memory is allocated to an object and how the program interprets the bit patterns found in the object's storage allocation. A data type is the set of values (often implementation-dependent) identifiers can assume, together with the set of operations allowed on those values.

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i;                    // declare i as an integer
unsigned char ch1, ch2;   // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++, a missing type specifier can lead to syntactic ambiguity, so C++ practice requires you to explicitly declare all **int** type specifiers

The type specifier keywords in Borland C++ are:

char	float	signed	wchar_t
class	int	struct	
double	long	union	
enum	short	unsigned	

Use the sizeof operators to find the size in bytes of any predefined or user-defined type.

asm, _asm, __asm

See also Example Keywords

Syntax

```
asm <opcode> <operands> <; or newline>
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description

Use the **asm, _asm**, or **_ _asm** keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the **asm** keyword, then surrounding the statements with braces ({ }). The initial brace must be on the same line as the **asm** keyword; placing it on the following line generates a syntax error.

Examples

```
// This example places a single assembler statement in your code:
asm pop dx

// If you want to include several of asm statements,
// surround them with braces:
asm {
  mov ax, 0x0e07
  xor bx, bx
  int 0x10          // makes the system beep
}
```

auto

Example Keywords

Syntax

```
[auto] <data-definition> ;
```

Description

Use the **auto** modifer to define a local variable as having a local lifetime.

This is the default for local variables and is rarely used.

Example

```
int main()
{
  auto int i;
  i = 5;
  return i;
}
```

break

See also Example Keywords

Syntax

```
break ;
```

Description

Use the **break** statement within loops to pass control to the first statement following the innermost enclosing brace.

Example

```
/* Illustrates the use of keywords break, case, default, and switch. */
#include <conio.h>
#include <stdio.h>

int main(void) {
   int ch;

   printf("\tPRESS a, b, OR c. ANY OTHER CHOICE WILL "
          "TERMINATE THIS PROGRAM.");
   for ( /* FOREVER */; ((ch = getch()) != EOF); )
      switch (ch) {
         case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
            printf("\nOption a was selected.\n");
            break;
         case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
         case 'c' :
            printf("\nOption b or c was selected.\n");
            break;
         default :
            printf("\nNOT A VALID CHOICE!  Bye ...");
            return(-1);
         }
   return(0);
   }
```

bool

See also Example Keywords

Syntax

bool <identifier>;

Description

Use **bool** and the literals **false** and **true** to make Boolean logic tests.

The **bool** keyword represents a type that can take only the value **false** or **true**. The keywords **false** and **true** are Boolean literals with predefined values. **false** is numericallly zero and **true** is numerically one. These Boolean literals are rvalues; you cannot make an assignment to them.

You can convert an rvalue that is **bool** type to an rvalue that is **int** type. The numerical conversion sets **false** to zero and **true** becomes one.

You can convert arithmetic, enumeration, pointer, or pointer to member rvalue types to an rvalue of type **bool**. A zero value, null pointer value, or null member pointer value is converted to **false**. Any other value is converted to **true**.

Example

```
/* How to make Boolean tests with bool, true, and false. */
#include <iostream.h>

bool func() {    // Function returns a bool type
   return NULL;  // NULL is converted to Boolean false
// return false;  // This statement is Boolean equivalent to the one above.
   }

int main() {
   bool val = false;  // Boolean variable
   int i = 1;         // i is neither Boolean-true nor Boolean-false
   int g = 3;
   int *iptr = 0;     // null pointer
   float j = 1.01;    // j is neither Boolean-true nor Boolean-false

   // Tests on integers
   if (i == true)  cout << "True: value is 1" << endl;
   if (i == false) cout << "False: value is 0" << endl;

   if (g) cout << "g is true.";
   else cout << "g is false.";

   // Test on pointer
   if (iptr == false) cout << "Invalid pointer." << endl;
   if (iptr == true) cout << "Valid pointer." << endl;

   // To test j's truth value, cast it to bool type.
   if (bool(j) == true) cout << "Boolean j is true." << endl;

   // Test Boolean function return value
   val = func();
   if (val == false)
      cout << "func() returned false.";
   if (val == true)
      cout << "func() returned true.";
   return false;  // false is converted to 0
}
```

Program output:

```
True: value is 1
Unknown truth value for g.
Invalid pointer.
Boolean j is true.
func() returned false.
```

case

See also Example Keywords

Syntax

```
switch ( <switch variable> ){
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the **case** statement in conjunction with switches to determine which statements evalute.

The list of possible branch points within `<statement>` is determined by preceding substatements with `case <constant expression> : <statement>;`

where `<constant expression>` must be an **int** and must be unique.

The `<constant expression>` values are searched for a match for the `<switch variable>`.

If a match is found, execution continues after the matching **case** statement until a **break** statement is encountered or the end of the **switch** statement is reached.

If no match is found, control is passed to the **default** case.

Note: It is illegal to have duplicate **case** constants in the same **switch** statement.

catch

See also Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the statements marked by the try keyword. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

cdecl, _cdecl, __cdecl

Example Keywords

Syntax

```
cdecl <data/function definition> ;
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a **cdecl**, **_cdecl**, or **__cdecl** modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use **cdecl**, **_cdecl**, or **__cdecl** in front of a function, it effects how the parameters are passed (last parameter is pushed first, and the caller cleans up the stack). The **__cdecl** modifier overrides the compiler directives and IDE options and allows the function to be called as a regular C function.

The **cdecl**, **_cdecl**, and **__cdecl** keywords are specific to Borland C++.

Example

```
int cdecl FileCount;
long far cdecl HisFunc(int x);
```

char

See also Keywords

Syntax

```
[signed|unsigned] char <variable_name>
```

Description

Use the type specifier **char** to define a character data type. Variables of type **char** are 1 byte in length.

A **char** can be signed, unsigned, or unspecified. By default, **signed char** is assumed.

Objects declared as characters (**char**) are large enough to store any member of the basic ASCII character set.

class

See also Example Keywords

Syntax

```
<classkey> <classname> [<:baselist>] { <member list> }
```

- `<classkey>` is either a class, struct, or union.
- `<classname>` can be any name unique within its scope.
- `<baselist>` lists the base class(es) that this class derives from. `<baselist>` is optional
- `<member list>` declares the class's data members and member functions.

Description

Use the **class** keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

Example

```
class stars {
   int magnitude;        // Data member
   int starfunc(void);   // Member function
};
```

const

See also Example Keywords

Syntax

```
const <variable name> [ = <value> ] ;
<function name> ( const <type>*<variable name> ;)
<function name> const;
```

Description

Use the **const** modifier to make a variable value unmodifiable.

Use the **const** modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a **const** result in a compiler error.

A **const** pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
   const float pi   = 3.14;
   const  maxint  = 12345;   // When used by itself, const is equivalent to
int.
   char *const str1 = "Hello, world";           // A constant pointer
   char const *str2 = "Borland International";  // A pointer to a constant
character string.
```

Given these declarations, the following statements are legal.

```
pi   = 3.0;         // Assigns a value to a const.
i     = maxint++;   // Increments a const.
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends **const** to include classes and member functions. In a C++ class definition, use the **const** modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the **const** keyword attempts to use only member functions that are also defined with **const**. If you call a member function that is not defined as **const**, the compiler issues a warning that the a non-**const** function is being called for a **const** object. Using the **const** keyword in this manner is a safety feature of C.

Warning: A pointer can indirectly modify a **const** variable, as in the following:

```
*(int *)&my_age = 35;
```

If you use the **const** modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

Example

```
class X  {
   int j;
public:
  X::X() { j = 0; };
  int lowerBound() const;            // DOES NOT MODIFY ANY DATA MEMBERS
  int dimension(X x1, const X &x2) { // x2 DATA MEMBERS WON'T BE MODIFIED
     x1.j = 3;       // OKAY; x1 OBJECT IS MODIFIABLE
     x2.j = 5;       // ERROR; x2 IS NOT MODIFIABLE
     return x2.j;
     }
};
```

Example 2

```
#include <iostream.h>

class Alpha {
   int num;
public:
   Alpha(int j = 0) { num = j; }
   int func(int i) const {
      cout << "Non-modifying function." << endl;
      return i++;
      }
   int func(int i) {
      cout << "Modify private data" << endl;
      return num = i;
      }
   int f(int i) { cout << "Non-const function called with i = " << i <<
  endl; return i;}
};

void main() {
   Alpha alpha_mod;         // Calls the non-const functions.
   const Alpha alpha_inst;  // Attempts to call the const functions.

   alpha_mod.func(1);
   alpha_mod.f(1);          // Causes a compiler warning.

   alpha_inst.func(1);
   alpha_inst.f(1);
   }
```

Output:

```
Modify private data
Non-const function called with i = 1
Non-modifying function.
Non-const function called with i = 1
```

continue

See also Example Keywords

Syntax

```
continue ;
```

Description

Use the **continue** statement within loops to pass control to the end of the innermost enclosing brace; at which point the loop continuation condition is re-evaluated.

Example

```
void main ()
{
  for (i = 0; i < 20; i++) {
     if (array[i] == 0)
       continue;
     array[i] = 1/array[i];
  }
}
```

_ _declspec

See also Example Keywords

Syntax

```
__declspec(decl-modifier)
```

Description

Use the **_ _declspec** keyword to indicate the storage class attributes for a DLL.

The **_ _declspec** keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The **_ _declspec** keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);                      // illegal
void __export f(void)                       // correct
void __declspec(dllexport) f(void);         // correct
__declspec(dllexport)void f(void);          // correct
class __declspec(dllexport) ClassName { }   // correct
```

The *decl-modifier* argument can only be one of *dllexport*, *dllimport*, or *thread*. The meaning of these arguments is equivalent to the following storage class attribute keywords.

Argument	Storage Class	Compiler Support
dllexport	__export	32- and 16-bit
dllimport	__import	32- bit (legal, but no affect on 16-bit programs)
thread	__thread	32- bit only

Example

```
/* Examples of  __declspec declarations follow. */
__declspec(dllimport) void func(void);
__declspec(dllimport) int a;
__declspec(dllexport) void bar (void);

/** Use thread argument only with static storage data. **/
__declspec(thread) int th;
int __declspec(thread) th1;
```

default

See also Example Keywords

Syntax

```
switch ( <switch variable> ){
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the default statement in switch statement blocks.

- If a case match is not found and the **default** statement is found within the switch statement, the execution continues at this point.
- If no default is defined in the **switch** statement, control passes to the next statement that follows the switch statement block.

operator delete

See also Example Operators

Syntax

```
<::> delete <cast-expression>
<::> delete [ ] <cast-expression>
delete <array-name> [ ];
```

Description

The **delete** operator offers dynamic storage deallocation, deallocating a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the **delete** operator to remove arrays that you no longer need. Failure to free memory can result in memory leaks.

The delete Operator with Arrays

Arrays are deleted by operator delete[](). You must use the syntax **delete []** *expr* when deleting an array. After C++ 2.1, the array dimension should not be specified within the brackets:

```
char * p;

void func()
{
   p = new char[10];     //  allocate 10 chars
   delete[] p;           // delete 10 chars
}
```

C++ 2.0 code required the array size. In order to allow 2.0 code to compile, Borland C++ issues a warning and simply ignores any size that is specified. For example, if the preceding example reads delete[10] p and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

Overloading the Operator delete

Example

The global operators, ::operator delete(), and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global delete function can exist in the program.

The user-defined operator delete must have a **void** return type and **void*** as its first argument; a second argument of type size_t is optional. A class *T* can define at most one version of each of *T*::**operator delete[]()** and *T*::**operator delete()**. To overload the **delete** operators, use the following prototypes.

- `void operator delete(void *Type_ptr, [size_t Type_size]);  // For Non-array`
- `void operator delete[](size_t Type_ptr, [size_t Type_size]);  // For arrays`

do

See also Example Keywords

Syntax

```
do <statement> while ( <condition> );
```

Description

The **do** statement executes until the condition becomes **false**.

`<statement>` is executed repeatedly as long as the value of `<condition>` remains **true**.

Since the conditon tests after each the loop executes the `<statement>`, the loop will execute at least once.

do Example

```
/* This example prompts users for a password  */
/* and continued to prompt them until they    */
/* enter one that matches the value stored in */
/* checkword.                                 */

#include <stdio.h>
#include <string.h>

int main ()
{
  char checkword[80] = "password";
  char password[80]  = "";

  do {
    printf ("Enter password: ");
    scanf("%s", password);
  } while (strcmp(password, checkword));

  return 0;
}
```

double

See also Keywords

Syntax

```
[long] double <identifier>
```

Description

Use the **double** type specifier to define an identifier to be a floating-point data type. The optional modifier **long** extends the accuracy of the floating-point value.

If you use the **double** keyword, the Borland C++ IDE will automatically link the floating-point math package into your program.

enum

See also Example Keywords

Syntax

```
enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];
```

- `<type_tag>` is an optional type tag that names the set.
- `<constant_name>` is the name of a constant that can optionally be assigned the value of `<value>`. These are also called enumeration constants.
- `<value>` must be an integer. If `<value>` is missing, it is assumed to be:

  `<prev> + 1`

  where `<prev>` is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
- `<var_list>` is an optional variable list that assigns variables to the enum type.

Description

Use the **enum** keyword to define a set of constants of type **int**, called an enumeration data type.

An enumeration data type provides mnemonic identifiers for a set of integer values. Borland C++ stores enumerators in a single byte if you uncheck Treat Enums As Ints (O|C|Code Generation) or use the **-b** flag.

Enums are always interpreted as **int**s if the range of values permits, but if they are not **int**s the value gets promoted to an **int** in expressions. Depending on the values of the enumerators, identifiers in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**.

In C, an enumerated variable can be assigned any value of type int--no type checking beyond that is enforced. In C++, an enumerated variable can be assigned only one of its enumerators.

In C++, lets you omit the **enum** keyword if `<tag_type>` is not the name of anything else in the same scope. You can also omit `<tag_type>` if no further variables of this **enum** type are required.

In the absence of a `<value>` the first enumerator is assigned the value of zero. Any subsequent names without initializers will then increase by one. `<value>` can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers.

In C++, enumerators declared within a class are in the scope of that class.

Examples

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, enum days, a variable anyday of this type, and a set of enumerators (sun, mon,...) with constant integer values.

```
enum modes { LASTMODE = -1, BW40=0, C40, BW80, C80, MONO = 7 };
/*
   "modes" is the type tag.
   "LASTMODE", "BW40", "C40", etc. are the constant names.
   The value of C40 is 1 (BW40 + 1); BW80 = 2 (C40 + 1), etc.
*/
```

__except

See also Keywords

Syntax

```
__except (expression) compound-statement
```

Description

The **__except** keyword specifies the action that should be taken when the exception specified by expression has been raised.

explicit

See also

Syntax

```
explicit <single-parameter constructor declaration>
```

Description

Normally, a class with a single-parameter constructor can be assigned a value that matches the constructor type. This value is automatically (implicitly) converted into an object of the class type to which it is being assigned. You can prevent this kind of implicit conversion from occurring by declaring the constructor of the class with the **explicit** keyword. Then all objects of that class must be assigned values that are of the class type; all other assignments result in a compiler error.

Objects of the following class can be assigned values that match the constructor type or the class type:

```
class X {
public:
  X(int);
  X(const char*, int = 0);
};
```

Then, the following assignment statements are legal.

```
void f(X arg) {
  X a = 1;
  X B = "Jessie";
  a = 2;
}
```

However, objects of the following class can be assigned values that match the class type only:

```
class X {
public:
  explicit X(int);
  explicit X(const char*, int = 0);
};
```

The **explicit** constructors then require the values in the following assignment statements to be converted to the class type to which they are being assigned.

```
void f(X arg) {
  X a = X(1);
  X b = X("Jessie",0);
  a = X(2);
}
```

_export, _ _export

See also Keywords

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with **_export** or **_ _export** into an export table for the module.

Using **_export** or **_ _export** eliminates the need for an EXPORTS section in your module definition file.

Note: Exported functions must be declared as **_ _far**. You can use the FAR type, defined in windows.h.

Functions that are not modified with **_export** or **_ _export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use **_export** or **_ _export** to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need a module definition file.

Prologs, Epilogs, and Exports: A Summary

See also Keywords

Prologs and epilogs are required when exporting functions in a 16-bit Windows application. They ensure that the correct data segment is active during callback functions and mark near and far stack frames for Windows stack crawling.

Two steps are required to export a function.

1. The compiler must create the correct prolog and epilog for the function.
2. The linker must create an entry for every export function in the header section of the executable.

In 32-bit Windows the binding of data segments does not apply. However, DLLs must have entries in the header so the loader can find the function to link to when an .EXE loads the DLL.

If a function is flagged with the **_ _export** keyword and any of the Windows compiler options are used, it will be compiled as exportable and linked as an export.

If a function is *not* flagged with the **_ _export** keyword, then one of the following situations will determine whether the function is exportable

- If you compile with the -tW/-tWC or -tWD/-tWCD option (or with the All Functions Exportable IDE equivalent), the function will be compiled as exportable.
- If the function is listed in the EXPORTS section of the module definition file, the function will be linked as an export. If it is not listed in the module definition file, or if no module definition file is linked, it won't be linked as an export.
- If you compile with the -tWE or -tWDE/-tWCDE option (or with the Explicit Functions Exported IDE equivalent), the function will *not* be compiled as exportable. Including this function in the EXPORTS section of the module definition will cause it be exported, but, because the prolog is incorrect, the program will run incorrectly. You may get a Windows error message in the 16-bit environment.

See the table, Compiler options and the _export keyword, for a summary of the effect of the combination of the Windows compiler options and the **_ _export** keyword.

Compiler Options and the _ _export Keyword

See also Keywords

This table summarizes the effect of the combination of various Windows options and the **_ _export** keyword:

The compiler option is: *	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**
Function flagged with __export?	Yes	Yes	Yes	Yes	No	No	No	No
Function·listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
Is·function exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes ****	**No *****	**No**

* Or the 32-bit console-mode application equivalents.

** The function will be exported in some sense, but because the prolog and epilog will not be correct, the function will not work as expected.

*** This combination also makes little sense. It is inefficient to compile all functions as exportable if you do not actually export some of them.

Smart Callbacks and the _export Keyword

See also Keywords

If you use the Smart Callbacks IDE option at compile time, callback functions do not need to be listed in the EXPORTS statement or flagged with the **_export** keyword.

Functions compile them so that they are callback functions.

Exportable Functions in DLLs

See also

There are two ways to compile a function `f1( )` in a DLL as exportable and then export it.

- Compile the DLL with all functions exportable (with the Windows DLL All Functions Exportable option in the IDE) and list `f1( )` in the EXPORTS section of the module definition file, or
- Flag the function `f1( )` with the _export keyword.

Using _export with C++ Classes

See also Keywords

Whenever you declare a class as **_export**, the compiler treats it as huge (with 32-bit pointers), and exports all of its non-inline member functions and static data members.

You cannot declare a class as **_export** and as **_far** or **_huge** (**_export** implies **_huge**, which implies **_far**).

If you declare the class in an include file that is included in both the DLL source files and the source files of the application that use the DLL, declare the class

- as **_export** when compiling the DLL
- as **_huge** when compiling the application

To do this, use the __DLL__ macro, which the compiler defines when it's building a DLL.

Note: In the mangled name, the compiler encodes the information that a given class member is a member of a huge class. This ensures that the linker will catch any mismatches when a program is using huge and non-huge classes.

Exporting and importing templates

See also

The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a DLL or an EXE file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use **_ _export** or **_ _import** modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used.

The following steps demonstrate exporting and importing of templates. The source code is organized in three files. Using the header file, code is generated in the DLL. A DLL library is created and linked to an EXE file.

1. Exportable/Importable Template Declarations

   The header file contains all template class and template function declarations. An export/import version of the templates can be instantiated by defining the appropriate macro at compile time.

2. Compiling Exportable Templates

   Write the source code for a DLL. When compiled, this DLL has reusable export code for templates.

3. Using ImportTemplates

   Now you can write a calling function that uses templates. This file is linked to the DLL. Only objects that are not declared in the header file and which are instantiated in the *main* function cause the compiler to generate new code. Code for a newly instantiated object is written into MAIN.OBJ file.

Using Import Templates

Program Output

```
// Before you compile this file you need to create the dynamic link library.
// You can use the command IMPLIB DLL_SRC.LIB DLL_SRC.DLL

// TO COMPILE THIS FILE, USE BCC32 -DUSING_DLL_IMPORTS MAIN DLL_SRC.LIB
#include <iostream.h>
#include "exporter.h"

int main () {
   int small = 5;
   int big = 10;
   double smalld = 1.2;
   double bigd = 12.3;

   // No new code is generated for these objects.
   Receive <double> Test_d(0.01);
   Receive <int> Test_i(5);

  // Generate code in MAIN.OBJ for this object.
  Receive <float> Test_f(3.14);
   cout << "Test_d.display() = " <<  Test_d.display() << endl;
   cout << "Test_i.display() = " <<  Test_i.display() << endl;

   cout << "min(5, 10): " << another_min(small, big) << endl;
   cout << "min(12.3, 1.2): " << another_min(bigd, smalld)<<endl;
   cout << "Test_f.display() = " <<  Test_f.display() << endl;

   return 0;
}
```

Program Output

```
Test_d.display() = 0.01
Test_i.display() = 5
min(5, 10): 5
min(12.3, 1.2): 1.2
Test_f.display() = 3.14
```

Compiling Exportable Templates

```
// In file DLL_SRC.CPP.
// GENERATE CODE FOR EXPORTABLE CLASSES AND FUNCTIONS.
// TO COMPILE THIS FILE, USE BCC32 -tWD -DBUILD_DLL_EXPORTS DLL_SRC.CPP
#define STRICT
#include <windows.h>
#include "exporter.h"

BOOL WINAPI DllEntryPoint(HINSTANCE hinstdll,
                          DWORD fdwReason, LPVOID lpvReserved)
{
  return 1;
}
```

Exportable/Importable Template Declarations

```
// In file EXPORTER.H
#include<iostream.h>
# if defined (BUILD_DLL_EXPORTS)
#     define DECLSPEC __export
# elif defined (USING_DLL_IMPORTS)
#     define DECLSPEC __import
# endif

////////////////////////////////////////////////
// Receive CLASS DEFINITIONS
template <class T> class Receive
{
  T value;
public:
  Receive(const T val) : value(val){}
   T display();
};

template<class T> T Receive<T>::display()
{
  return value;
}

// TEMPLATE FUNCTION DEFINITION
template <class T>
T another_min(T a, T b) { return a < b ? a : b;}

#if (defined (BUILD_DLL_EXPORTS) ||  defined(USING_DLL_IMPORTS) )
////// INSTANTIATED TEMPLATE CLASSES /////
template class DECLSPEC Receive<double>;
template class DECLSPEC Receive<int>;
template class DECLSPEC Receive<char>;

////// INSTANTIATED TEMPLATE FUNCTIONS /////
template int DECLSPEC another_min<int>(int, int);
template double DECLSPEC another_min<double>(double, double);
#endif
```

extern

See also Example Keywords

Syntax

```
extern <data definition> ;
[extern] <function prototype> ;
```

Description

Use the **extern** modifier to indicate that the actual storage and initial value of a variable, or body of a function, is defined in a separate source code module. Functions declared with extern are visible throughout all source files in a program, unless you redefine the function as static.

The keyword **extern** is optional for a function prototype.

Use `extern "c"` to prevent function names from being mangled in C++ programs.

Examples

```
extern int _fmode;
extern void Factorial(int n);
extern "c" void cfunc(int);
```

far, _far, __far

Example Keywords

Syntax

```
<type> far <pointer definition> ;
<type> far <function definition>
<type> _far <pointer definition> ;
<type> _far <function definition>
<type> __far <pointer definition> ;
<type> __far <function definition>
```

Description

Use the **far**, **_far** and **__far** modifiers to generate function code for calls and returns using variables that are outside of the data segment.

The first version of **far**, **_far**, or **__far** declares a pointer to be two words with a range of 1 megabyte. Use **_ _far** when compiling small or compact models to force pointers to be **_ _far**.

Examples

```
char __far *s;
void * __far * p;
int __far my_func() {}
```

_fastcall, __fastcall

See also Keywords

Syntax

```
return-value _fastcall function-name(parm-list)
return-value __fastcall function-name(parm-list)
```

Description

Use the **_fastcall** modifiers to declare functions that expect parameters to be passed in registers.

The compiler treats this calling convention as a new language specifier, along the lines of _cdecl and _pascal

Functions declared using **_cdecl** or **_pascal** cannot also have the **_fastcall** modifiers because they use the stack to pass parameters. Likewise, the **_fastcall** modifiers cannot be used together with _export or _loadds.

The compiler generates a warning if you mix functions of these types or if you use the **_fastcall** modifiers in a dangerous situation. You can, however, use functions that use the **_fastcall** or **__fastcall** conventions in overlaid modules (for example, with modules that will use VROOMM).

The compiler prefixes the **_fastcall** function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

Note: The **__fastcall** modifier is subject to name mangling. See the description of the -VC option.

Parameter Types and Possible Registers Used

The compiler uses the following rules when deciding which parameters are to be passed in registers.

Parameter Type	Registers
char (signed and unsigned)	AL, DL, BL
int (signed and unsigned)	AX, DX, BX
long (signed and unsigned)	DX:AX
near pointer	AX, DX, BX

Only three parameters can be passed in registers to any one function.

Do not assume the assignment of registers will reflect the ordering of the parameters to a function. Far pointer, union, structure, and floating-point (**float**, **double**, and **long**) parameters are pushed on the stack.

__finally

See also Keywords

Syntax

```
__finally {compound-statement}
```

Description

The **__finally** keyword specifies actions that should be taken regardless of how the flow within the preceding __try exits.

The **__finally** keyword is supported only in C programs.

float

See also Keywords

Syntax

```
float <identifier>
```

Description

Use the **float** type specifier to define an identifier to be a floating-point data type.

Type	Length	Range
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)

The Borland C++ IDE automatically links the floating-point math package into you program if you use floating-point values or operators.

for

Example Keywords

Syntax

`for ( [<initialization>] ; [<condition>] ; [<increment>] )  <statement>`

Description

The **for** statement implements an iterative loop.

`<statement>` is executed repeatedly UNTIL the value of `<condition>` is false.

- Before the first iteration of the loop, `<initialization>` initializes variables for the loop.
- After each iteration of the loop, `<increments>` increments a loop counter. Consequently, `j++` is functionally the same as `++j`.

In C++, `<initialization>` can be an expression or a declaration.

The scope of any identifier declared within the **for** loop extends to the end of the control statement only.

A variable defined in the **for**-*initialization* expression is in scope only within the **for**-block. See the description of the -Vd option.

All the expressions are optional. If `<condition>` is left out, it is assumed to be always true.

Examples

```
// An example of the scope of variables in for-expressions.
// The example compiles if you use the -Vd option.
#include <iostream.h>

int main() {
   for (int i = 0; i < 10; i++)
      if (i == 8)
      cout << "\ni = " << i;
return i;  // Undefined symbol 'i' in function main().
}
```

friend

Example Keywords

Syntax

```
friend <identifier>;
```

Description

Use **friend** to declare a function or class with full access rights to the private and protected members of an outside class, without being a member of that class.

In all other respects, the **friend** is a normal function in terms of scope, declarations, and definitions.

Example

```
class stars {
   friend galaxy;
   int magnitude;
   int starfunc(void);
};

class galaxy {
   long int number_of_stars;
   void stars_magnitude(stars&);
   void stars_func(stars*);
}
```

goto

Example Keywords

Syntax

```
goto <identifier> ;
```

Description

Use the **goto** statement to transfer control to the location of a local label specified by <identifier>.

Labels are always terminated by a colon.

Example

```
Again:       /* this is the label */
;
.
.
.
goto Again;
```

huge, _huge, _ _huge

See also Keywords

Syntax

```
<type> huge <pointer-definition> ;
<type> _huge <pointer-definition> ;
<type> __huge <pointer-definition> ;
```

Description

The **_huge** modifiers are similar to the **_far** modifier except for two additional features.

- Its segment is normalized during pointer arithmetic so that pointer comparisons are accurate.
- Huge pointers can be incremented without suffering from segment wraparound.

if

See also Example Keywords

Syntax

```
if ( <condition> )  <statement1>;

if ( <condition> )  <statement1>;
else  <statement2>;
```

Description

Use **if** to implement a conditional statement.

You can declare variables in the condition expression. For example,

```
if (int val = func(arg))
```

is valid syntax. The variable *val* is in scope for the **if** statement and extends to an **else** block when it exists.

The condition statement must convert to a **bool** type. Otherwise, the condition is ill-formed.

When `<condition>` evaluates to **true**, `<statement1>` executes.

If `<condition>` is **false**, `<statement2>` executes.

The **else** keyword is optional, but no statements can come between an **if** statement and an **else**.

The #if and #else preprocessor statements (directives) look similar to the **if** and **else** statements, but have very different effects. They control which source file lines are compiled and which are ignored.

Examples

```
if (int val = func(count)) { /* statements */ }
else {
   /* take other action */
   cout << "val is false"
   }
```

_import, _ _import

Keywords

Form 1

```
class _import <class name>
class __import <class name>
```

Form 2

```
return_type _import <function name> //32-bit only
return_type __import <function name> //32-bit only
```

Form 3

```
data_type _import <data name> //32-bit only
data_type __import <data name> //32-bit only
```

Description

This keyword can be used as a class modifier for 16-bit programs; and as a class, function, or data modifier in 32-bit programs. If you're importing classes that are declared with the modifier __huge, you must change the modifier to the keyword **_ _import**. The **_ _huge** modifier merely causes far addressing of the virtual tables (the same effect as the -Vf compiler option). The **_ _import** modifier makes all function and static addresses default to _far.

inline

Example Keywords

Syntax

```
inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **inline** keyword to declare or define C++ inline functions.

Inline functions are best reserved for small, frequently used functions.

Example

```
inline char* cat_func(void) { return char*; }
```

int

See also Keywords

Syntax

```
[signed|unsigned] int <identifier> ;
```

Description

Use the **int** type specifier to define an integer data type.

Variables of type **int** can be signed (default) or unsigned.

_ _interrupt functions

Example Keywords

Syntax

```
interrupt <function-definition> ;
_interrupt <function-definition> ;
__interrupt <function-definition> ;
```

Description

Use the **_ _interrupt** function modifier to define a function as an interrupt handler. This keyword is only available in BCC.EXE for use in 16-bit applications.

The **__interrupt** modifier is specific to Borland C++. **__interrupt** functions are designed to be used with interrupt vectors.

Interrupt functions compile with extra function entry and exit code so that all CPU registers are saved. The BP, SP, SS, CS, and IP registers are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function uses an IRET instruction to return, so that the function can be used as harware and software interrupts.

Declare interrupt functions to be of type **void** and can be declared in any memory model. For all memory models except huge, DS is set to the program data segment. For the huge memory model, DS is set to the module's data segment.

Example

```
void interrupt myhandler()
{
  ...
}
```

_loadds, __loadds

Keywords

Syntax

```
_loadds <function-name>
__loadds <function-name>
```

Description

Use the **_ _loadds** keyword to indicate that a function should set the DS register, just as a huge function does.

These keywords are useful for writing low-level interface routines, such as mouse support routines.

long

See also Keywords

Syntax

```
long [int] <identifier> ;
[long] double <identifier> ;
```

Description

When used to modify an **int**, it doubles the number of bytes available to store the integer value.

When used to modify a **double**, it defines a floating-point data type with 80 bits of precision instead of 64.

The Borland C++ IDE links the floating-point math package if you use floating-point values or operators anywhere in your program.

near, _near, _ _near

Example Keywords

Syntax

```
<type> near <pointer definition> ;
<type> near <function definition>
<type> _near <pointer definition> ;
<type> _near <function definition>
<type> __near <pointer definition> ;
<type> __near <function definition>
```

Description

Use **near** and **_near** type modifiers to force pointers to be near and to generate function code for a near call and a near return.

The first version of **_near** declares a pointer to be one word with a range of 64K.

Use this type modifier when compiling in the medium, large, or huge memory models to force pointers to be near.

When either **near** or **_near** is used with a function declaration, the compiler generates function code for a **near** call and a **near** return.

Example

```
char near *s;
int (near *ip)[10];
int near my_func() {}
```

operator new

See also Example Operators

Syntax

```
<::> new <placement> type-name <(initializer)>
<::> new <placement> (type-name) <(initializer)>
```

Description

The **new** operator offers dynamic storage allocation, similar but superior to the standard library function malloc. The **new** operator must always be supplied with a data type in place of *type-name*. Items surrounded by angle brackets are optional. The optional arguments can be as follows:

- **::** operator, invokes the global version of **new**.
- *placement* can be used to supply additional arguments to **new**. You can use this syntax only if you have have an overloaded version of **new** that matches the optional arguments. See the discussion of the placement syntax.
- *initializer*, if present is used to initialize the allocation. Arrays cannot be initialized by the allocation operator.

A request for non-array allocation uses the appropriate **operator new()** function. Any request for array allocation will call the appropriate **operator new[]()** function. The selection of the allocation operator is done as follows:

Allocation of arrays of *Type*:

1 Attempts to use a class-specific array allocator:

Type::**operator new[]()**

2 If the class-specific array allocator is not defined, the global version is used:

::operator new[]()

Allocation of non-arrays of *Type*:

1 Attempts to used the class-specific allocator:

Type::**operator new()**

2 If the class-specific array allocator is not defined, the global version is used:

::operator new()

Allocation of single objects (that are not class-type) which are not held in arrays:

1 Memory allocation for a non-array object is by using the **::operator new()**. Note that this allocation function is always used for the predefined types. It is possible to overload this global operator function. However, this is generally not advised.

Allocation of arrays:

1 Use the global allocation operator:

::operator new[] ()

Note: Arrays of classes require the default constructor.

new tries to create an object of type *Type* by allocating (if possible) sizeof(*Type*) bytes in free store (also called the heap). new calculates the size of *Type* without the need for an explicit **sizeof** operator. Further, the pointer returned is of the correct type, "pointer to *Type*," without the need for explicit casting. The storage duration of the **new** object is from the point of creation until the operator **delete** destroys it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the allocated memory. By default, an allocation failure (such as insufficient or fragmented heap memory) results in the predefined exception xalloc being thrown. Your program should always be prepared to catch the *xalloc* exception before trying to access the new object (unless you use a new-handler).

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

Operator new placement syntax

Example

The *placement* syntax for **operator new( )** can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the *placement* syntax when you want to use and reuse a memory space which you set up once at the beginning of your program.

When you use the overloaded **operator new( )** to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global **::operator delete( )** to do the cleanup.

To release memory, you make an explicit call on the destructor. This method for cleaning up memory should be used only in special situations and with great care. If you make an explicit call of a destructor before an object that has been constructed on the stack goes out of scope, the destructor will be called again when the stackframe is cleaned up.

operator new placement syntax example

```
// An example of the placement syntax for operator new()
#include <iostream.h>

class Alpha {
  union {
    char  ch;
    char  buf[10];
    };
public:
   Alpha(char c = '\0') : ch(c) {
      cout << "character constructor" << endl;
      }
   Alpha(char *s) {
      cout << "string constructor" << endl;
      strcpy(buf,s);
      }

   ~Alpha( ) { cout << "Alpha::~Alpha() " << endl; }

   void * operator new(size_t, void * buf) {
      return buf;
      }
};

void main() {
   char *str = new char[sizeof(Alpha)];

   // Place 'X' at start of str.
   Alpha* ptr = new(str) Alpha('X');
   cout << "str[0] = " << str[0] << endl;

   // Explicit call of the destructor
   ptr -> Alpha::~Alpha();

   // Place a string in str buffer.
   ptr = new(str) Alpha("my string");
   cout << "\n str = " << str << endl;

   // Explicit call of the destructor
   ptr -> Alpha::~Alpha();
   delete[] str;
   }
```

Output:

```
character constructor
str[0] = X
Alpha::~Alpha()
string constructor

 str = my string
Alpha::~Alpha()
```

Handling Errors for the new Operator

See also

You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use `set_new_handler(0)` .

The Operator new With Arrays

Example

If Type is an array, the pointer returned by operator new[]() points to the first element of the array. When creating multidimensional arrays with new, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];     // OK
mat_ptr = new int[n][10][12];     // OK
mat_ptr = new int[3][][12];       // illegal
mat_ptr = new int[][10][12];      // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

// Example of the new and delete Operators

```
// ALLOCATE A TWO-DIMENSIONAL SPACE, INITIALIZE, AND DELETE IT.
#include <except.h>
#include <iostream.h>

void display(long double **);
void de_allocate(long double **);

int m = 3;                                  // THE NUMBER OF ROWS.
int n = 5;                                  // THE NUMBER OF COLUMNS.

int main(void) {
   long double **data;

   try {                                    // TEST FOR EXCEPTIONS.
      data = new long double*[m];           // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < m; j++)
          data[j] = new long double[n];     // STEP 2: SET UP THE COLUMNS
      }
   catch (xalloc) {  // ENTER THIS BLOCK ONLY IF xalloc IS THROWN.
      // YOU COULD REQUEST OTHER ACTIONS BEFORE TERMINATING
      cout << "Could not allocate. Bye ...";
      exit(-1);
      }

   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
          data[i][j] = i + j;               // ARBITRARY INITIALIZATION

   display(data);
   de_allocate(data);
   return 0;
   }

void display(long double **data) {
   for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
             cout << data[i][j] << " ";
       cout << "\n" << endl;
       }
   }

void de_allocate(long double **data) {
   for (int i = 0; i < m;  i++)
       delete[] data[i];                    // STEP 1: DELETE THE COLUMNS

   delete[] data;                           // STEP 2: DELETE THE ROWS
   }
```

operator new

See also

By default, if there is no overloaded version of new, a request for dynamic memory allocation always uses the global version of new, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type name, a specific operator called *name*::**operator new()** or *name*::**operator new[]()** can be defined. When **new** is applied to class name objects it invokes the appropriate *name*::**operator new** if it is present; otherwise, the global **::operator new** is used.

Only the **operator new()** function will accept an optional initializer. The array allocator version, **operator new[](),** will not accept initializers. In the absence of explicit initializers, the object created by new contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor. The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

Overloading the operator new

Example

The global ::operator new() and ::operator new[]() can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific memory allocation operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a size_t as its first argument. To overload the **new** operators, use the following prototypes declared in the new.h header file.

- void * operator new(size_t Type_size); // For Non-array
- void * operator new[](size_t Type_size); // For arrays

The Borland C++ compiler provides *Type_size* to the **new** operator. Any data type may be substitued for *Type_size* except function names (although a pointer to function is permitted), class declarations, enumeration declarations, const, volatile.

Example of Overloading the new and delete Operators

```
#include <stdlib.h>

class X {
   .
   .
   .
public:
   void* operator new(size_t size) { return newalloc(size);}
   void operator delete(void* p) { newfree(p); }
   X() { /* initialize here */ }
   X(char ch) { /* and here */ }

   ~X() { /* clean up here */ }
   .
   .
   .
};
```

The *size* argument gives the size of the object being created, and *newalloc* and *newfree* are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of class *X* (or objects of classes derived from *X* that do not have their own overloaded operators **new** and **delete**) will invoke the matching user-defined *X*::**operator new()** and *X*::**operator delete()**, respectively. (Destructors will be called only if you use the **-xd** compiler option and an exception is thrown.)

The *X*::**operator new()**, *X*::**operator new[]()**, *X*::**operator delete()** and *X*::**operator delete[]()** operator functions are static members of *X* whether explicitly declared as static or not, so they cannot be virtual functions.

The standard, predefined (global) **::operator new()**, **::operator new[]()**, **:: operator delete()**, and **::operator delete[]()** operators can still be used within the scope of *X*, either explicitly with the global scope or implicitly when creating and destroying non-*X* or non-*X*-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
   void* ptr = new char[s]; // standard new called
      .
      .
      .
   return ptr;
}

void X::operator delete(void* ptr)
{
      .
      .
      .
   delete (void*) ptr;     // standard delete called
}
```

The reason for the *size* argument is that classes derived from *X* inherit the *X*::**operator new()** and *X*::**operator new[]()**. The size of a derived class object may well differ from that of the base class.

operator

See also Example Keywords

Syntax

```
operator <operator symbol>( <parameters> )
{
   <statements>;
}
```

Description

Use the **operator** keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

Example

```
new_complex operator +(complex c1, complex c2)
{
   return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

pascal, _pascal, __pascal

Example Keywords

Syntax

```
pascal <data-definition/function-definition> ;
_pascal <data-definition/function-definition> ;
__pascal <data-definition/function-definition> ;
```

Description

Use the **pascal**, **_pascal**, and **__pascal** keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, **pascal** declares Pascal-style parameter-passing conventions when applied to a function header (first parameter pushed first; the called function cleans up the stack).

In C++ programs, functions declared with the **pascal** modifer will still be mangled.

Examples

```
int pascal FileCount;
far pascal long ThisFunc(int x, char *s);
```

private

See also Keywords

Syntax

```
private: <declarations>
```

Description

A private member can be accessed only by member functions and friends of the class in which it is declared.

Class members are **private** by default.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

protected

See also Keywords

Syntax

```
protected: <declarations>
```

Description

A protected member can be accessed by member functions and friends of the class in which it was declared, and by classes derived from the declared class.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

public

See also Keywords

Syntax

```
public: <declarations>
```

Description

A public member can be accessed by any function.

Members of a struct or union are public by default.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

register

Example Keywords

Syntax

```
register <data definition> ;
```

Description

Use the **register** storage class specifier to store the variable being declared in a CPU register (if possible), to optimize access and reduce code.

Items declared with the **register** keyword have a global lifetime.

Note: The Borland C++ compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Example

```
register int i;
```

return

Example Keywords

Syntax

```
return [ <expression> ] ;
```

Description

Use the return statement to exit from the current function back to the calling routine, optionally returning a value.

Example

```
double sqr(double x)
{
  return (x*x);
}
```

_saveregs, _ _saveregs

See also Keywords

Syntax

```
_saveregs <function-name>;
__saveregs <function-name>;
```

Description

The **__saveregs** modifier causes the function to preserve all register values and restore them before returning (except for explicit return values passed in registers such as AX or DX). **__saveregs** is not available in flat mode.

Use this keyword with functions; it is useful for writing low-level interface routines, such as mouse support routines.

Note: The **__saveregs** modifier is subject to name mangling. See the description of the -VC option.

_seg, _ _seg

Example Keywords

Syntax

```
<datatype> _seg *<identifier> ;
<datatype> __seg *<identifier> ;
```

Description

Use _seg in 16-bit segment pointer type declarators. _seg is not available in flat mode.

Any indirection through `<identifier>` has an assumed offset of 0. In arithmetic involving segment pointers they are treated like pointers except for the following restictions.

1. You cannot use the ++, --, +=, or -= operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. If you add a near pointer to a segment pointer, the operation creates a far pointer result by using the segment from the segment pointer and the offset from the near pointer.

   Therefore, the two pointers must point to the same type, or one must be a pointer to void.

   There is no multiplication of the offset, regardless of the type pointed to.
4. When a segment pointer is used in an indirection expression, it also implicitly converts to a far pointer.
5. If you add or subtract an integer operand to or from a segment pointer, the result is a far pointer. The segment is taken from the segment pointer; the offset is calculated by multiplying the size of the object pointed to by the integer operand.
6. Segment pointers can be assigned, initialized, passed into and out of functions, and compared.

Example

```
int _seg *name;
```

signed

See also Example Keywords

Syntax

```
signed <type> <variable> ;
```

Description

Use the signed type modifier when the variable value can be either positive or negative. The signed modifier can be applied to base types **int**, **char**, **long** and **short**.

When the base type is omitted from a declaration, int is assumed.

Example

```
signed   int       i;    /* signed is default        */
signed             i;    /* same as "signed int i;"  */
unsigned long int  l;    /* int OK, not needed       */
signed   char      ch;   /* unsigned is default      */
```

short

See also Example Keywords

Syntax

```
short int <variable> ;
```

Description

Use the short type modifier when you want a variable smaller than an int. This modifier can be applied to the base type int.

When the base type is omitted from a declaration, int is assumed.

Examples

```
short int i;
short     i;    /* same as "short int i;" */
```

The sizeof operator

See also Example Operators

The **sizeof** operator has two distinct uses:

sizeof *unary-expression*

sizeof (*type-name*)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals `sizeof array/ sizeof array[0]`.

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

You cannot use **sizeof** with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is *size_t*, defined in stddef.h.

You can use **sizeof** in preprocessor directives; this is specific to Borland C++.

In C++, `sizeof(classtype),` where *classtype* is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example for sizeof operator

```
/*  USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
   char *name;     /* 2 BYTES IN SMALL-DATA MODELS; 4 BYTES IN LARGE-DATA
  MODEL */
   int age;        /* 2 BYTES IN SMALL-DATA MODELS; 4 BYTES IN LARGE-DATA
  MODEL */
   double height;  /* ALWAYS EIGHT BYTES */
      };

struct st St_Array[]= {  /* AN ARRAY OF structs */
   { "Jr.",     4,  34.20 },  /* ST_Array[0] */
   { "Suzie",  23,  69.75 },  /* ST_Array[1] */
   };
int main() {
   long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };

   printf("("\nNumber of elements in LD_Array = %d",
           sizeof(LD_Array) / sizeof(LD_Array[0]));

   /****  THE NUMBER OF ELEMENTS IN THE ST_Array. ****/
   printf("\nSt_Array has %d elements",
           sizeof(St_Array)/sizeof(St_Array[0]));

   /****  THE NUMBER OF BYTES IN EACH ST_Array ELEMENT.  ****/
   printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));

   /****  THE TOTAL NUMBER OF BYTES IN ST_Array.  ****/
      printf("\nSt_Array=%d", sizeof(St_Array));
   return 0;
   }
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 12
St_Array= 24
```

static

Example Keywords

Syntax

```
static <data definition> ;
static <function name> <function definition> ;
```

Description

Use the **static** storage class specifier with a local variable to preserve the last value between successive calls to that function. A **static** variable acts like a local variable but has the lifetime of an external variable.

In a class, data and member functions can be declared **static**. Only one copy of the **static** data exists for all objects of the class.

A **static** member function of a global class has external linkage. A member of a local class has no linkage. A **static** member function is associated only with the class in which it is declared. Therefore, such member functions cannot be **virtual**.

Static member functions can only call other **static** member functions and only have access to **static** data. Such member functions do not have a **this** pointer.

Examples

```
static int i;
static void printnewline(void) {}
```

_stdcall, __stdcall

Keywords

Syntax

```
__stdcall <function-name>
_stdcall <function-name>
```

Description

The **_stdcall** and **__stdcall** keywords force the compiler to generate function calls using the Standard calling convention.The resulting function calls are smaller and faster. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. Such functions comply with the standard WIN32 argument-passing convention.

Note: The **__stdcall** modifier is subject to name mangling. See the description of the -VC option.

struct

See also Example Keywords

Syntax

```
struct [<struct type name>] {
  [<type> <variable-name[, variable-name, ...]>] ;
    .
    .
    .
} [<structure variables>] ;
```

Description

Use a **struct** to group variables into a single record.

`<struct type name>`	An optional tag name that refers to the structure type.
`<structure variables>`	The data definitions, also optional.

Though both `<struct type name>` and `<structure variables>` are optional, one of the two must appear.

You define elements in the record by naming a `<type>`, followed by one or more `<variable-name>` (separated by commas).

Separate different variable types by a semicolon.

To access elements in a structure, use a record selector (.).

To declare additional variables of the same type, use the keyword **struct** followed by the `<struct type name>`, followed by the variable names.

Note: Borland C++ allows the use of anonymous struct embedded within another structure.

Example

```
struct my_struct {
  char name[80], phone_number[80];
  int  age, height;
} my_friend;

strcpy(my_friend.name,"Mr. Wizard");   /* accessing an element */

struct my_struct my_friends[100];   /* declaring additional variables */
```

switch

See also Example Keywords

Syntax

```
switch ( <switch variable> ) {
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the switch statement to pass control to a case which matches the `<switch variable>`. At which point the statements following the matching case evaluate .

If no case satisfies the condition the default case evaluates.

To avoid evaluating any other cases and reliquish control from the switch, terminate each case with `break;`.

Example

```
/* Illustrates the use of keywords break, case, default, and switch. */
#include <conio.h>
#include <stdio.h>

int main(void) {
   int ch;

   printf("\tPRESS a, b, OR c. ANY OTHER CHOICE WILL "
          "TERMINATE THIS PROGRAM.");
   for ( /* FOREVER */; ((ch = getch()) != EOF); )
      switch (ch) {
         case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
            printf("\nOption a was selected.\n");
            break;
         case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
         case 'c' :
            printf("\nOption b or c was selected.\n");
            break;
         default :
            printf("\nNOT A VALID CHOICE!  Bye ...");
            return(-1);
         }
   return(0);
   }
```

template

See also Keywords

Syntax

```
template-declaration:
   template < template-argument-list > declaration

template-argument-list:
   template-argument
   template-argument-list, template argument

template-argument:
   type-argument
   argument-declaration

type-argument:
   class identifier

template-class-name:
   template-name < template-arg-list >

template-arg-list:
   template-arg
   template-arg-list , template-arg

template-arg:
   expression
   type-name

< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

this

See also Keywords

Syntax

```
class X {
  int a;
public:
  X (int b) {this -> a = b;}
```

Description

In nonstatic member functions, the keyword **this** is a pointer to the object for which the function is called. All calls to nonstatic member functions pass **this** as a hidden argument.

this is a local variable available in the body of any nonstatic member function. Use it implicitly within the function for member references. It does not need to be declared and it is rarely referred to explicitly in a function definition.

For example, in the call `x.func(y)` , where y is a member of X, the keyword **this** is set to &x and y is set to `this->y`, which is equivalent to x.y.

Static member functions do not have a **this** pointer because they are called with no particular object in mind. Thus, a static member function cannot access nonstatic members without explicitly specifying an object with . or ->.

throw

See also Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type *T* (to match the type of argument *arg*) used in `throw(T arg)`. Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

_ _try

See also Keywords

Syntax

```
_ _try compound-statement handler-list
_ _try compound-statement termination-statement
```

Description

The **_ _try** keyword is supported only in C programs. Use try in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword **__try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

// try example

```
// In PROG.C
void func(void) {
   // generate an exception
   RaiseException( // specify your arguments );
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
   try
   {            // test for C++ exceptions
      try
      {         // test for C-based structured exceptions
         func();
      }
      __except( /* filter-expression */ )
      {
      cout << "A structured exception was generated.";
      /* specify actions to take for this structured exception */
      return -1;
      }
      return 0;
   }
   catch ( ... )
   {
   // handler for any C++ exception
   cout << "A C++ exception was thrown.";
   return 1;
   }
}
```

try

See also Example Keywords

Syntax

```
try compound-statement handler-list
```

Description

The **try** keyword is supported only in C++ programs. Use __try in C programs.

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is tranferred to the handler

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

typedef

Example Keywords

Syntax

```
typedef <type definition> <identifier> ;
```

Description

Use the **typedef** keyword to assign the symbol name `<identifier>` to the data type definition `<type definition>`.

typename

Syntax 1

```
typename <identifier>
```

Syntax 2

```
template <  typename <identifier>  > class <identifier>
```

Description

Use the syntax 1 to reference a type that you have not yet defined. See example 1.

Use syntax 2 in place of the **class** keyword in a template declaration. See example 2.

typename Example 2

```
/* This example shows how the typename keyword can be used to replace the
  class keyword in a template declaration. */

#include <iostream.h>

template <typename T1, typename T2> T2 convert (T1 t1)
                              // use typename instead of class.
{ return (T2)t1; }

template <typename X, class Y> bool isequal (X x, Y y)
                              // mix typename and class.
{ if (x==y)return 1; return 0; }
```

typename Example 1

```
/* This example uses the typename keyword to declare variables as type T::A,
  which has not yet been defined. */

template <class T>
void f() {
  typedef typename T::A TA;   // declare TA as type T::A
  TA a5;                      // declare a5 as type TA
  typename T::A a6;           // declare a6 as type T::A
  TA * pta6;                  // declare pta6 as pointer to type TA
}
```

Examples

```
typedef unsigned char byte;
typedef char str40[41];
typedef struct {
     double re, im;
   } complex;
```

The typeid operator

See also Example Keywords

Syntax

```
typeid( expression )
typeid( type-name )
```

Description

You can use **typeid** to get run-time identification of types and expressions. A call to **typeid** returns a reference to an object of type **const** typeinfo. The returned object represents the type of the **typeid** operand.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

If the **typeid** operand is a dereferenced NULL pointer, the *Bad_typeid* exception is thrown.

// typeid example

```
// HOW TO USE operator typeid, Type_info::before(), AND Type_info::name()
#include <iostream.h>
#include <typeinfo.h>

class A { };
class B : A { };

void main() {
   char C;
   float X;

   // USE THE typeinfo::operator==()TO MAKE COMPARISON
   if (typeid( C ) == typeid( X ))
      cout << "C and X are the same type." << endl;
   else cout << "C and X are NOT the same type." << endl;

   // USE true AND false LITERALS TO MAKE COMPARISON
   cout << typeid(int).name();
   cout << " before " << typeid(double).name() << ": " <<
        (typeid(int).before(typeid(double)) ? true : false) << endl;
   cout << typeid(double).name();

   cout << " before " << typeid(int).name() << ": " <<
        (typeid(double).before(typeid(int)) ? true : false) << endl;

   cout << typeid(A).name();
   cout << " before " << typeid(B).name() << ": " <<
        (typeid(A).before(typeid(B)) ? true : false) << endl;
   }
```

Program Output

```
C and X are NOT the same type.
int before double: 0
double before int: 1
A before B: 1
```

union

See also Example Keywords

Syntax

```
union [<union type name>] {
  <type> <variable names> ;
  ...
} [<union variables>] ;
```

Description

Use unions to define variables that share storage space.

The compiler allocates enough storage in a_number to accommodate the largest element in the union.

Unlike a struct, the variables a_number.i and a_number.l occupy the same location in memory. Thus, writing into one overwrites the other.

Use the record selector (.) to access elements of a union .

Example

```
union int_or_long {
  int     i;
  long    l;
} a_number;
```

unsigned

See also Example Keywords

Syntax

```
unsigned <type> <variable> ;
```

Description

Use the **unsigned** type modifier when variable values will always be positive. The **unsigned** modifer can be applied to base types **int**, **char**, **long,** and **short**.

When the base type is omitted from a declaration, **int** is assumed.

Examples

```
unsigned int       i;
unsigned           i;    /* same as "unsigned int i;"    */
unsigned long int  l;    /* int OK, not needed           */
unsigned char      ch;   /* unsigned is default for char */
```

virtual

See also Keywords

Syntax

```
virtual class-name
virtual function-name
```

Description

Use the **virtual** keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as **virtual**, you can redefine it in any derived class, even if the number and type of arguments are the same.

The redefined function overrides the base class function.

void

Example Keywords

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the **void** keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
  printf("Hello, %s.",name);
}
```

Use **void** as a function heading if the function does not take any parameters.

```
int init(void)
{
  return 1;
}
```

Void Pointers

Generic pointers can also be declared as **void**, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

Example

```
int x;
float r;
void *p = &x;          /* p points to x */
int main (void)

  *(int *) p = 2;
  p = &r;              /* p points to r */
  *(float *)p = 1.1;
}
```

volatile

See also Keywords

Syntax

```
volatile <data definition> ;
```

Description

Use the **volatile** modifier to indicate that a variable can be changed by a background routine, an interrupt routine, or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a register variable

```
volatile int ticks;
void _ _interrupt timer( ) {
   ticks++;
}
void wait (int interval) {
   ticks = 0;
   while (ticks < interval);  // Do nothing
}
```

The routines in this example (assuming *timer* has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop since the loop doesn't change the value of *ticks*.

Note: C++ extends **volatile** to include classes and member functions. If you've declared a **volatile** object, you can use only its **volatile** member functions.

while

See also Example Keywords

Syntax

```
while ( <condition> ) <statement>
```

Description

Use the **while** keyword to conditionally iterate a statement.

`<statement>` executes repeatedly until the value of `<condition>` is **false**. If no condition is specified, the **while** clause is equivalent to **while**(**true**).

The test takes place before `<statement>` executes. Thus, if `<condition>` evaluates to **false** on the first pass, the loop does not execute.

Example

```
while (*p == ' ') p++;
```

Data Types (16-bit)

See also Keywords

Type	Length	Range
unsigned char	8 bits	0to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	64 bits	1.7×10^{-308} to $1.7 \times 10^{+308}$
long double	80 bits	3.4×10^{-4932} to $1.1 \times 10^{+4932}$
near (pointer)	16 bits	not applicable
far (pointer)	32 bits	not applicable

Data Types (32-bit)

See also Keywords

Type	Length	Range
unsigned char	8 bits	0to 255
char	8 bits	-128 to 127
short int	16 bits	-32,768 to 32,767
unsigned int	32 bits	0 to 4,294,967,295
int	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	32 bits	0 to 4,294,967,295
enum	16 bits	-2,147,483,648 to 2,147,483,647
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	64 bits	1.7×10^{-308} to $1.7 \times 10^{+308}$
long double	80 bits	3.4×10^{-4932} to $1.1 \times 10^{+4932}$
near (pointer)	32 bits	not applicable
far (pointer)	32 bits	not applicable

_cs, __cs, _ds, __ds, _ss, __ss, _es, __es

Example Keywords

Syntax

```
<type> _cs <pointer definition> ;
<type> __cs <pointer definition> ;
<type> _ds <pointer definition> ;
<type> __ds <pointer definition> ;
<type> _ss <pointer definition> ;
<type> __ss <pointer definition> ;
<type> _es <pointer definition> ;
<type> __es <pointer definition> ;
```

Description

Use the _cs, __cs, _ds, __ds, _ss, __ss, _es, and __es keywords to define special versions of near data pointers.

These pointers are 16-bit offsets associated with the specified segment register: CS, DS, SS or ES.

Example

```
char _cs  *s;     /* in cs code segment  */
int  _ss  ix ;    /* in ss stack segment */
long _ds  l[4] ;  /* in ds data segment  */
char _es  m[8] ;  /* in es segment       */
```

Table of Borland C++ register pseudovariables

Example

_AH	**_CL**	**_EAX**[1]	**_ESP**
_AL	**_CS**	**_EBP**[1]	**_FLAGS**
_AX	**_CX**	**_EBX**[1]	**_FS**[1]
_BH	**_DH**	**_ECX**[1]	**_GS**[1]
_BL	**_DI**	**_EDI**[1]	**_SI**
_BP	**_DL**	**_EDX**[1]	**_SP**
_BX	**_DS**	**_ES**	**_SS**
_CH	**_DX**	**_ESI**[1]	

1 These pseudovariables are always available to the 32-bit compiler. The 16-bit compiler can use these only when you use the option to generate 80386 instructions.

All but the _FLAGS register pseudovariable are associated with the general purpose, segment, address, and special purpose registers.

Use register pseudovariables anywhere that you can use an integer variable to directly access the corresponding 80x86 register.

The 16-bit flags register contains information about the state of the 80x86 and the results of recent instructions.

Example

```
_AX = 0x4c00;
```

_ _rtti and the -RT option

See also Example Keywords

RTTI is enabled by default in Borland C++. You can use the **-RT** command-line option to disable it (**-RT-**) or to enable it (**-RT**). If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class, typeid returns a reference to a **const** *typeinfo* object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic run-time type identification (where appropriate) by using the Borland C++ keyword **_ _rtti** in the class definition.

When you use the **-RT-** compiler option, if any base class is declared **_ _rtti**, then all polymorphic base classes must also be declared **_ _rtti**.

```
struct __rtti S1 { virtual s1func(); };  /* Polymorphic */
struct __rtti S2 { virtual s2func(); };  /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism (by using the **-RT-** compiler option), RTTI might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations with -RT-, you should declare *X* with the **_ _rtti** modifier. Otherwise, switching the order of the base classes for the class *X* results in the compile-time error Can't inherit non-RTTI class from RTTI base '*S1*'.

```
struct _ _rtti S1 { virtual func(); };   /* Polymorphic class */
struct S2 { };                           /*  Non-polymorphic class */
struct _ _rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with **_ _rtti**. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared **_ _rtti**.

```
struct _ _rtti S1 {  };   // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { };   // The order is not essential
```

Applying either **_ _rtti** or using the -RT compiler option will not make a static class into a polymorphic class.

-RT option and destructors

When **-xd** is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with **-RT**. The **-RT** and **-xd** options are on by default.

Example

```
class Alpha {
public:
   virtual ~Alpha( ) { }
};
void func( Alpha *Aptr ) {
   delete Aptr;          // Error.  Alpha is not a polymorphic class type
   }
```

Run-time type identification (RTTI) overview

See also

Run-time type identification (RTTI) lets you write portable code that can determine the actual type of a data object at run time even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the dynamic_cast operator to make run-time casts.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with typeid operator, which determines the actual type of its argument and returns a reference to an object of type **const** *typeinfo*, which describes that type.

You can also use a type name as the argument to **typeid**, and **typeid** will return a reference to a **const** *typeinfo* object for that type. The class typeinfo provides an **operator==** and an **operator!=** that you can use to determine whether two objects are of the same type. Class *typeinfo* also provides a member function name that returns a pointer to a character string that holds the name of the type.

_ _rtti Example

```
/* HOW TO GET RUN-TIME TYPE INFORMATION FOR POLYMORPHIC CLASSES.*/
#include <iostream.h>
#include <typeinfo.h>

class __rtti Alpha {        /* Provide RTTI for this class and */
                             /* all classes derived from it */

    virtual void func() {};  /* A virtual function makes */
                             /* Alpha a polymorphic class. */
};

class B : public Alpha {};

int main(void) {
   B Binst;           // Instantiate class B
   B *Bptr;           // Declare a B-type pointer
   Bptr = &Binst;     // Initialize the pointer

   // THESE TESTS ARE DONE AT RUN TIME
   try {
      if (typeid( *Bptr ) == typeid( B ) )
          // Ask "WHAT IS THE TYPE FOR *Bptr?"
          cout << "Name is " << typeid( *Bptr).name();
      if (typeid( *Bptr ) != typeid( Alpha ) )
          cout << "\nPointer is not an Alpha-type.";
      return 0;
      }
   catch (Bad_typeid) {
      cout << "typeid() has failed.";
      return 1;
      }
   }
```

Program Output

```
Name is B
Pointer is not an Alpha-type.
```

wchar_t (keyword)

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, **wchar_t** is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. A **wchar_t** type is the same size, signedness, and alignment requirement as an **int** type.

defined

See also Keywords

Syntax

```
#if defined[(] <identifier> [)]
#elif defined[(] <identifier> [)]
```

Description

Use the defined operator to test if an identifier was previously defined using #define. The **defined** operator is only valid in #if and #elif expressions.

Defined evaluates to 1 (true) if a previously defined symbol has not been undefined (using #undef); otherwise, it evaluates to 0 (false).

Defined performs the same function as #ifdef.

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the #if directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

mutable

See also Example Keywords

Syntax

```
mutable <variable name>;
```

Description

Use the **mutable** specifier to make a variable modifiable even though it is in a **const**-qualified expression.

Using the mutable Keyword

Only class data members can be declared mutable. The **mutable** keyword cannot be used on **static** or **const** names. The purpose of **mutable** is to specify which data members can be modified by **const** member functions. Normally, a **const** member function cannot modify data members.

Example

```
#include <iostream.h>
class Alpha {
   mutable int count;
   mutable const int* iptr;
public:
   int func1(int i = 0) const { // Promises not to change const arguments.
      count = i++;  // But count can be changed.
      iptr = &i;
      cout << *iptr;
      return count;
      }
};

int main(void) {
   Alpha a;

   a.func1(0);
   return 0;
   }
```

Namespaces overview

See Also

Most nontrivial applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided by the C++ namespace mechanism.

The namespace mechanism allows an application to be partitioned into number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.

It only takes two steps to use C++ namespaces. The first is to uniquely identify a namespace with the keyword **namespace**. The second is to access the elements of an identified namespace by applying the **using** keyword.

Defining a namespace

SeeAlso

The grammar for defining a namespace is

original-namespace-name:
identifier

namespace-definition:
original-namespace-definition
extension-namespace-definition
unnamed-namespace-definition

Grammatically, there are three ways to define a namespace with the namespace keyword:

original-namespace-definition:
namespace *identifier* { *namespace-body* }

extension-namespace-definition:
namespace *original-namespace-name* { *namespace-body* }

unnamed-namespace-definition:
namespace { *namespace-body* }

The body is an optional sequence of declarations. The grammar is

namespace-body:
declaration-seq opt

Example

```
// An example of the using directive
#include <iostream.h>
   namespace F {
      float x = 9;
      }
   namespace G {
      using namespace F;
      float y = 2.0;
         namespace INNER_G {
            float z = 10.01;
            }
      }

   int main() {
      using namespace G;  // THIS DIRECTIVE GIVES YOU EVERYTHING DECLARED IN
  "G"
      using namespace G::INNER_G;  // THIS DIRECTIVE GIVES YOU ONLY
  "INNER_G"
      float x = 19.1;     // LOCAL DECLARATION TAKES PRECEDENCE
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << "z = " << z << endl;
      return 0;
      }
```

Output:

```
   x = 19.1
   y = 2
   z = 10.01
```

Declaring a namespace

SeeAlso

An original namespace declaration should use an identifier that has not been previously used as a global identifier.

```
namespace ALPHA {  /* ALPHA is the identifier of this namespace. */
  /* your program declarations */
  long double LD;
  float f(float y) { return y; }
  }
```

A namespace identifier must be known in all translation units where you intend to access it's elements.

Namespace alias

You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.

```
namespace BORLAND_INTERNATIONAL {
   /* namespace-body */
   namespace NESTED_BORLAND_INTERNATIONAL {
      /* namespace-body */
      }
   }

// Alias namespace
namespace BI = BORLAND_INTERNATIONAL;

// Use access qualifier to alias a nested namespace
namespace NBI = BORLAND_INTERNATIONAL::NESTED_BORLAND_INTERNATIONAL;
```

Extending a namespace

Example

Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.

Example for extending namespaces

```
// An example for extending namespaces
#include <iostream.h>
   struct S { };
   class C  { };

   namespace ALPHA {  // ALPHA is an original identifier.
      void g(struct S) {
         cout << "Processing a structure argument" << endl;
         }
      }

   using ALPHA::g;  // using-declaration

   /*** After the using-declaration above, subsequent attempts
        to overload the g() function are ignored. ***/
   namespace ALPHA { // Extending the ALPHA namespace
      void g( C& ) { // Overloaded version of function
         cout << "Processing a class argument." << endl;
         }
      }

   int main() {
      S mystruct;
      C myclass;

      g(mystruct);

      // The following function call fails at compile-time
      // because there is no overloaded version for this case.
//      g(myclass);
      return 0;
   }
```

Output:

```
Processing a structure argument
```

Anonymous namespaces

Example

The C++ grammar allows you to define anonymous namespaces. To do this,you use the keyword **namespace** with no identifier before the enclosing brace.

```
namespace {           // Anonymous namespace
   // Declarations
   }
```

All anonymous, unnamed namespaces in global scope (that is, unnamed namespaces that are not nested) of the same translation unit share the same namespace. This way you can make static declarations without using the **static** keyword.

Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.

Example

In file ANON1.CPP

```
#include <iostream.h>
extern void func(void);

namespace {          // Anonymous
   float pi = 3.14;  // Unique identifier known only in this file
   }

void main() {
   float pi = 0.1;
   cout << "pi = " << pi << endl;
   func();
   }
```

In file ANON2.CPP

```
#include <iostream.h>

namespace {             // Anonymous namespace
   float pi = 10.0001;  // Unique identifier known only in this file
   void func(void) {
       cout << "First func() called; pi = " << pi;
       }
   }
   void func(void) {
       cout << "Second func() called; pi = " << pi;
       }
```

Program output:

```
pi = 0.1
func() called; pi = 10.0001
```

Accessing elements of a namespace

There are three ways to access the elements of a namespace: by explicit access qualification, the using-declaration, or the using-directive. Remember that no matter which namespace you add to your local scope, identifiers in global scope (global scope is just another namespace) are still accessible by using the scope resolution operator **::**.

Explicit access qualification

Using directive

Using declaration

Accessing namespaces in classes

Example

You cannot use a **using** directive inside a class. However, the **using** declarative is allowed and can be quite useful.

Using directive

Example

If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive specifies that all identifiers in a namespace are in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows.

using-directive:

using namespace **::** opt *nested-name-specifier* opt *namespace-name*;

The using-directive is transitive. That means that when you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your program, you also get namespaces *A*, *ONE*, and *TWO*.

```
namespace A {
   using namespace ONE;  // This has been defined previously
   using namespace TWO;  // This also has been defined previously
   }
```

The using-directive does not add any identifiers to your local scope. Therefore, an identifier defined in more than one namespace won't be a problem until you actually attempt to use it. Local scope declarations take precedence by hiding all other similar declarations.

Using declaration

Example

You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is

using-declaration:

using :: *unqualified-identifier*;

Example

```
// An example of the using declaration.
// The function g() is defined in two different namespaces.
#include <iostream.h>

namespace ALPHA {  /* ALPHA is the name of this namespace. */
   float f(float y) { return y; }
   void g() { cout << "ALPHA version" << endl; }
   }
namespace BETA {  /* BETA is the name of this namespace. */
      void g() { cout << "BETA version" << endl; }
         }

void main(void) {
// The using declaration identifies the desired version of g().
         using ALPHA::f;  // Qualified declaration
         using BETA::g;   // Qualified declaration
         float x = 0;

         // Access qualifiers are no longer needed.
         x = f(2.1);
         g();
         }
```

Explicit access qualification

You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the **::** scope resolution operator followed by the member name. For example, to access a specific member of namespace *ALPHA*, you write:

```
ALPHA::LD;  // Access a variable
ALPHA::f;   // Access a function
```

Explicit access qualification can always be used to resolve ambiguity. No matter which namespace (except anonymous namespace) is being used in your subsystem, you can apply the scope resolution operator **::** to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace. Therefore, any identifier in the application can be accessed with sufficient qualification.

Example

```
// An example for accessing a namespace within a class.
// This allows us to overload a function which is a base class member.

#include <iostream.h>
   class A {
   public:
      void func(char ch) { cout << "char = " << ch << endl; }
   };

   class B : public A {
   public:
//       using namespace A;   // ERROR. The using directive isn't allowed
      void func(char *str) { cout << "string = " << str << endl; }

      // The using declarative
      using A::func;         // Overload B::func()
   };

   int main() {
      B b;

      b.func('c');  // Calls A::func()
      b.func("c");  // Calls B::func()
      return 0;
   }
```

Operators Summary

See also

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

Arithmetic

Assignment

Bitwise

C++ specific

Comma

Conditional

Equality

Logical

Postfix Expression Operators

Primary Expression Operators

Preprocessor

Reference/Indirect operators

Relational

sizeof

typeid

All operators can be overloaded except the following:

`.`	C++ direct component selector
`.*`	C++ dereference
`::`	C++ scope access/resolution
`?:`	Conditional

Depending on context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as:

- a bitwise AND (A & B)
- an address operator (&A)
- in C++, a reference modifier

Note: No spaces are allowed in compound operators. Spaces change the meaning of the operator and will generate an error.

Associativity and Precedence of Operators

Operators

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the same line have equal precedence.

Operators											Associativity
()	[]	->	::	.							left to right
!	~	+	-	++	--	&	*	sizeof	new	delete	right to left
.*	->*										left to right
*	/	%									left to right
+	-										left to right
<<	>>										left to right
<	<=	>	>=								left to right
==	!=										left to right
&											left to right
^											left to right
|											left to right
&&											left to right
||											right to left
?:											left to right
=	*=	/=	%=	+=	-=	&=	^=	|=	<<=	>>=	right to left
,											left to right

Arithmetic Operators

See also Operators

Syntax

```
+ cast-expression
- cast-expression
add-expression + multiplicative-expression
add-expression - multiplicative-expression
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

Remarks

Use the arithmetic operators to perform mathematical computations.

The unary expressions of + and - assign a positive or negative value to the cast-expression.

+ (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types, integer and floating point.

% (modulus operator) returns the remainder of integer division and cannot be used with floating points.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Assignment Operators

See also Operators

Syntax

```
unary-expr  assignment-op  assignment-expr
```

Remarks

The assignment operators are:

```
=     *=      /=      %=     +=     -=
<<=   >>=     &=      ^=     |=
```

The = operator is the only simple assignment operator, the others are compound assignment operators.

In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op= E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue E1 is evaluated only once. For example, E1 += E2 is the same as E1 = E1 + E2.

The expression's value is E1 after the expression evaluates.

For both simple and compound assignment, the operands E1 and E2 must obey one of the following rules:

1. E1 is a qualified or unqualified arithmetic type and E2 is an arithmetic type.
2. E1 has a qualified or unqualified version of a structure or union type compatible with the type of E2.
3. E1 and E2 are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
4. Either E1 or E2 is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
5. E1 is a pointer and E2 is a null pointer constant.

Note: Spaces separating compound operators (+<space>=) will generate errors.

Bitwise operators

See also Operators

Syntax

```
AND-expression &  equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
~cast-expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Remarks

Use the bitwise operators to modify the individual bits rather than the number.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
|	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. ~ is used to create destructors.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and assigns the left most bit to 0.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns the right most bit to 0.

Both operands in a bitwise expression must be of an integral type.

Bit value		Results of		
E1	**E2**	**E1 & E2**	**E1 ^ E2**	**E1 | E2**
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive. & can also be the pointer reference operator.

>> can also be the input operator in I/O expressions.

<< can also be the output operator in I/O expressions.

C++ Specific Operators

See also Operators

The operators specific to C++ are:

::	Scope access (or resolution) operator
.*	Dereference pointers to class members
->*	Dereference pointers to pointers to class members
const_cast	adds or removes the **const** or **volatile** modifier from a type
delete	dynamically deallocates memory
dynamic_cast	converts a pointer to a desired type
new	dynamically allocates memory
reinterpret_cast	replaces casts for conversions that are unsafe or implementation dependent.
static_cast	converts a pointer to a desired type
typeid	gets run-time identification of types and expressions

Use the scope access (or resolution) operator ::(two semicolons) to access a global (or file duration) name even if it is hidden by a local redeclaration of that name.

Use the .* and ->* operators to dereference pointers to class members and pointers to pointers to class members.

Comma Punctuator and Operator

See also Operators

Syntax

```
expression , assignment-expression
```

Remarks

The comma separates elements in a function argument list.

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

The left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls func with three arguments (i, 5, k), not four.

Conditional Operator

Operators

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Remarks

The conditional operator ?: is a ternary operator.

In the expression E1 ? E2 : E3, E1 evaluates first. If its value is **true**, then E2 evaluates and E3 is ignored. If E1 evaluates to **false**, then E3 evaluates and E2 is ignored.

The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which evaluates.

E1 must be a scalar expression. E2 and E3 must obey one of the following rules:

1. Both of arithmetic type. E2 and E3 are subject to the usual arithmetic conversions, which determines the resulting type.
2. Both of compatible **struct** or **union** types. The resulting type is the structure or union type of E2 and E3.
3. Both of **void** type. The resulting type is **void**.
4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
6. One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**. The resulting type is that of the non-pointer-to-**void** operand.

Logical Operators

Operators

Syntax

```
logical-AND-expr  && inclusive-OR-expression
logical-OR-expr   || logical-AND-expression
! cast-expression
```

Remarks

Operands in a logical expression must be of scalar type.

&&	logical AND; returns **true** only if both expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **false**, the second expression is not evaluated.
||	logical OR; returns **true** if either of the expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **true**, the second expression is not evaluated.
!	logical negation; returns **true** if the entire expression evaluates to be nonzero, otherwise returns **false**. The expression !E is equivalent to (0 == E).

Primary Expression Operators

For ANSI C, the primary expressions are *literal* (also sometimes referred to as *constant*), identifier, and (*expression*). The C++ language extends this list of primary expressions to include the keyword **this**, scope resolution operator **::**, *name*, and the class destructor **~** (tilde).

The primary expressions are summarized in the following list.

primary-expression:

literal
this (C++ specific)
:: *identifier* (C++ specific)
:: *operator-function-name* (C++ specific)
:: *qualified-name* (C++ specific)
(expression)
name

literal:

integer-constant
character-constant
floating-constant
string-literal

name:

identifier
perator-function-name (C++ specific)
conversion-function-name (C++ specific)
~ *class-name* (C++ specific)
qualified-name (C++ specific)

qualified-name: (C++ specific)

qualified-class-name **::** *name*

For a discussion of the primary expression **this**, see this (keyword). The keyword **this** cannot be used outside a class member function body.

The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The parenthesis surrounding an *expression* do not change the unadorned expression itself.

The primary expression *name* is restricted to the category of primary expressions that sometimes appear after the member access operators **.** (dot) and **–>** . Therefore, name must be either an lvalue or a function. See also the discussion of member access operators.

An *identifier* is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown in Lexical Elements: Identifiers.

See the discussion on how to use the destructor operator ~ (tilde).

Postfix expression operators

See also Operators

Syntax

```
postfix-expression(<arg-expression-list>)
array declaration [constant-expression]
compound statement { statement list }
postfix-expression . identifier
postfix-expression -> identifier
```

Remarks

( )	use to group expressions, isolate conditional expressions, indicate function calls and function parameters
{ }	use as the start and end of compound statements
[ ]	use to indicate single and multidimensional array subscripts
.	use to access structure and union members
->	use to access structure and union members

The following postfix expressions let you make safe, explicit typecasts in a C++ program.

const_cast< T > (expression)

dynamic_cast< T > (expression)

reinterpret_cast< T > (expression)

static_cast< T > (expression)

To obtain run-time type identification (RTTI), use the **typeid()** operator. The syntax is as follows:

typeid(expression)

typeid(type-name)

Preprocessor Operators

Operators

Remarks

The # (pound sign) is a preprocessor directive when it occurs as the first non-whitespace character on a line.

It signifies a compiler action, not necessarily associated with code generation.

and ## (double pound signs) also perform operator replacement and merging during the preprocessor scanning phase.

Reference/Dereference Operators

See also Operators

Syntax

```
& cast-expression
* cast-expression
```

Remarks

The **&** and ***** operators work together to reference and dereference pointers that are passed to functions.

Referencing operator (&)

Use the reference operator to pass the address of a pointer to a function outside of *main().*

The cast-expression operand must be one of the following:

- a function designator
- an lvalue designating an object that is not a bit field and is not declared with a register storage class specifier

If the operand is of type <*type*>, the result is of type pointer to <*type*>.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer-to-X" types when they appear in certain contexts. The **&** operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following example:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;      // Initialized pointer
*ptr = t2;           // Same effect as t1 = t2
```

`T *ptr = &t1` is treated as

```
T *ptr;
ptr = &t1;
```

So it is *ptr*, or **ptr*, that gets assigned. Once *ptr* has been initialized with the address *&t1*, it can be safely dereferenced to give the lvalue **ptr*.

Indirection operator (*)

Use the asterisk (*****) in a variable expression to create pointers. And use the indirect operator in external functions to get a pointer's value that was passed by reference.

If the operand is of type *pointer to function*, the result is a function designator.

If the operand is a pointer to an object, the result is an lvalue designating that object.

The result of indirection is undefined if either of the following occur:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Note: & can also be the bitwise AND operator.

* can also be the multiplication operator.

Relational Operators

See also Operators

Syntax

```
relational-expression  <   shift-expression
relational-expression  >   shift-expression
relational-expression  <=  shift-expression
relational-expression  >=  shift-expression
```

Remarks

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be **true** it returns a nonzero character; otherwise it returns **false** (0).

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

In the expression

```
E1 <operator> E2
```

the operands must follow one of these conditions:

1. Both E1 and E2 are of arithmetic type.
2. Both E1 and E2 are pointers to qualified or unqualified versions of compatible types.
3. One of E1 and E2 is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void.
4. One of E1 or E2 is a pointer and the other is a null pointer constant.

Array subscript operator

See also Operators

Brackets ([ ]) indicate single and multidimensional array subscripts. The expression

```
<exp1>[exp2]
```

is defined as

```
*((exp1) + (exp2))
```

where either:

- exp1 is a pointer and exp2 is an integer or
- exp1 is an integer and exp2 is a pointer

Function call operator

See also Operators

Syntax

```
postfix-expression(<arg-expression-list>)
```

Remarks

Parentheses ()

- group expressions
- isolate conditional expressions
- indicate function calls and function parameters

The value of the function call expression, if it has a value, is determined by the return statement in the function definition.

This is a call to the function given by the postfix expression.

arg-expression-list is a comma-delimited list of expressions of any type representing the actual (or real) function arguments.

Direct member selector

See also Example Operators

Syntax

```
postfix-expression . identifier
```

postfix-expression must be of type union or structure.

identifier must be the name of a member of that structure or union type.

Remarks

Use the selection operator (.) to access structure and union members.

Suppose that the object *s* is of struct type S and *sptr* is a pointer to S. Then, if *m* is a member identifier of type M declared in S, this expression:

```
s.m
```

are of type M, and represent the member object *m* in *s*.

Example

```
struct mystruct {
   int i
   char str[21]
   double d
} s, *sptr=&s
 ...
s.i = 3          // assign to the i member of mystruct s
```

The expression s.m is an lvalue, provided that s is not an lvalue and m is not an array type.

If structure B contains a field whose type is structure A, the members of A can be accessed by two applications of the member selectors.

Indirect member selector

See also Example Operators

Syntax

```
postfix-expression -> identifier
```

postfix-expression must be of type pointer to structure or pointer to union.

identifier must be the name of a member of that structure or union type.

The expression designates a member of a structure or union object. The value of the expression is the value of the selected member it will be an lvalue if and only if the postfix expression is an lvalue.

Remarks

You use the selection operator -> to access structure and union members.

Suppose that the object *s* is of struct type S and *sptr* is a pointer to S. Then, if *m* is a member identifier of type M declared in S, this expression:

```
sptr->m
```

is of type M, and represents the member object *m* in *s*.

The expression

```
s->sptr
```

is a convenient synonym for (*sptr).m.

-> Example

```
struct mystruct {
   int i
   char str[21]
   double d
} s, *sptr=&s
  .
  .
  .
sptr->d = 1.23   // assign to the d member of mystruct s
```

The expression sptr->m is an lvalue unless m is an array type.

If structure B contains a field whose type is structure A, the members of A can be accessed by two applications of the member selectors.

Increment/Decrement operators

Operators

Increment operator (++)

Syntax

```
postfix-expression ++        (postincrement)
++ unary-expression          (preincrement)
```

The expression is called the operand it must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue..

Postincrement operator

The value of the whole expression is the value of the postfix expression before the increment is applied.

After the postfix expression is evaluated,the operand is incremented by 1.

Preincrement operator

The operand is incremented by 1 before the expression is evaluated the value of the whole expression is the incremented value of the operand.

The increment value is appropriate to the type of the operand.

Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

Syntax

```
postfix-expression --        (postdecrement)
-- unary-expression          (predecrement)
```

The decrement operator follows the same rules as the increment operator, except that the operand is decremented by 1 after or before the whole expression is evaluated.

Plus and Minus Operators

See also Operators

Unary

In these unary + - expressions

```
+ cast-expression
- cast-expression
```

the cast-expression operand must be of arithmetic type.

Results

+ cast-expression	Value of the operand after any required integral promotions.
- cast-expression	Negative of the value of the operand after any required integral promotions.

Binary

Syntax

```
add-expression + multiplicative-expression
add-expression - multiplicative-expression
```

Legal operand types for op1 + op2:

1. Both op1 and op2 are of arithmetic type.
2. op1 is of integral type, and op2 is of pointer to object type.
3. op2 is of integral type, and op1 is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Legal operand types for op1 - op2:

1. Both op1 and op2 are of arithmetic type.
2. Both op1 and op2 are pointers to compatible object types.
3. op1 is of pointer to object type, and op2 is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Note: The unqualified type <type> is considered to be compatible with the qualified types const type, volatile type,and const volatile type.

Multiplicative Operators

See also Operators

Syntax

```
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
```

Remarks

There are three multiplicative operators:

* (multiplication)

/ (division)

% (modulus or remainder)

The usual arithmetic conversions are made on the operands.

(op1 * op2) Product of the two operands

(op1 / op2) Quotient of (op1 divided by op2)

(op1 % op2) Remainder of (op1 divided by op2)

For / and %, op2 must be nonzero op2 = 0 results in an error. (You can't divide by zero.)

When op1 and op2 are integers and the quotient is not an integer:

1. If op1 and op2 have the same sign, op1 / op2 is the largest integer less than the true quotient, and op1 % op2 has the sign of op1.
2. If op1 and op2 have opposite signs, op1 / op2 is the smallest integer greater than the true quotient, and op1 % op2 has the sign of op1.

Note: Rounding is always toward zero.

* is context sensitive and can be used as the pointer reference operator.

Punctuators

See also Operators

The Borland C++ punctuators (also known as separators) are:

( ) Parentheses

{ } Braces

, Comma

; Semicolon

: Colon

... Ellipsis

* Asterisk

= Equal Sign

Pound Sign

Most of these punctuators also function as operators.

Braces

Operators

The braces ({ }) indicate the start and end of a compound statement.

Semicolon

Operators

The semicolon (;) is a statement terminator.

Any legal C expression (including the empty expression) followed by is interpreted as a statement, known as an expression statement.

The expression is evaluated and its value is discarded. If the expression statement has no side effects, Borland C++ can ignore it.

Semicolons are often used to create an empty statement.

Colon

Example Operators

Use the colon (:) to indicate a labeled statement:

Example

```
start:    x=0
...
goto start
...
switch (a) {
    case 1: puts("One")
            break
    case 2: puts("Two")
            break
 ...
 default:   puts("None of the above!")
            break
}
```

Ellipsis

Operators

An ellipsis (...) is three successive periods with no intervening whitespace.

Use an ellipsis in formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types. For example,

```
void func(int n, char ch,...)
```

This declaration indicates that calls to *func* must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.

The comma preceding the ellipsis is not necessary.

Equal sign

Operators

The equal sign (=) separates variable declarations from initialization lists.

```
char array[5] = { 1, 2, 3, 4, 5 } ;
int  x = 5;
```

In a C function, no code can precede any variable declarations.

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }  //  parameter i has default value of zero
```

The equal sign is also used as the assignment operator.

Binary operators

See also Operators

These are the binary operators in Borland C++:

Arithmetic	+	Binary plus (add)
	-	Binary minus (subtract)
	*	Multiply
	/	Divide
	%	Remainder (modulus)
Bitwise	<<	Shift left
	>>	Shift right
	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	|	Bitwise inclusive OR
Logical	&&	Logical AND
	||	Logical OR
Assignment	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
Relational	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	==	Equal to
	!=	Not equal to
Component selection	.	Direct component selector
	->	Indirect component selector
Class-member	::	Scope access/resolution
	.*	Dereference pointer to class member
	->*	Dereference pointer to class member
Conditional	? :	Actually a ternary operator for example,
	a ? x : y	"if a then x else y"
Comma	,	Evaluate

Unary operators

See also Operators

Syntax

```
<unary-operator> <unary expression>
```

OR

```
<unary-operator> <type><unary expression>
```

Remarks

Unary operators group right-to-left.

Borland C++ provides the following unary operators:

! Logical negation

* Indirection

++ Increment

~ Bitwise complement

-- Decrement

- Unary minus

+ Unary plus

Overloading Operators

See also Operators

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators can be overloaded except for:

`.  .*  ::  ?:`

The followoing preprocessing symbols cannot be overloaded.

`#  ##`

The =, [ ], ( ), and -> operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for **enum** types. Any attempt to overload a global version of these operators results in a compile-time error.

The keyword **operator** followed by the operator symbol is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.

Example for Overloading Operators

The following example extends the class complex to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax.

Some of the issues illustrated by the example are:

- The default constructor is defined. This is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of *cvector* types, you specify exactly the actions to be taken. Note that the assignment operator function cannot be inherited by derived classes
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The **const** version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for *cvector* types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a **friend**. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.
- The stream insertion operator is overloaded to naturally display a *cvector*. Large objects that don't display well on a limited size screen might require a different display strategy.

Source

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
#pragma warn -inl     // IGNORE not expanded inline WARNINGS.
#include <complex.h>  // THIS ALREADY INCLUDES iostream.h
// COMPLEX VECTORS
class cvector {
   int size;
   complex *data;
public:
   cvector() { size = 0; data = NULL; };
   cvector(int i = 5) : size(i) {    // DEFAULT VECTOR SIZE.
      data = new complex[size];
      for (int j = 0; j < size; j++)
           data[j] = j + (0.1 * j);  // ARBITRARY INITIALIZATION.
      };
   /* THIS VERSION IS CALLED IN main() */
   complex& operator [](int i) { return data[i]; };
   /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR
  */
   const complex& operator [](int i) const { return data[i]; };
   cvector operator +(cvector& A) {  // ADDITION OPERATOR
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < size; i++)
          result[i] = data[i] + A.data[i];
      return result;
      };
   /* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER O
  F
      THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENS
```

```
  URE
      PROPER MULTIPLICATION. */
   friend cvector operator *(int scalar, cvector& A) {
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < A.size; i++)
        result.data[i] = scalar * A.data[i];
      return result;
      }
   /* THE STREAM INSERTION OPERATOR. */
   friend ostream& operator <<(ostream& out_data, cvector& C) {
      for (int i = 0; i < C.size; i++)
           out_data << "[" << i << "]=" << C.data[i] << "   ";
      cout << endl;
      return out_data;
      };
   cvector( const cvector &C ) {  // COPY CONSTRUCTOR
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
           data[i] = C[i];
     }
   cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
      if (this == &C) return *this;
      delete[] data;
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
          data[i] = C[i];
      return *this;
   };
   virtual ~cvector() { delete[] data; }; // DESTRUCTOR
   };
int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
   cvector cvector1(4), cvector2(4), result(4);
   // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
   cvector1[3] = complex(3.3, 102.8);
   cout << "Here is cvector1:" << endl;
   cout << cvector1;
   cvector2[3] = complex(33.3, 81);
   cout << "Here is cvector2:" << endl;
   cout << cvector2;
   result = cvector1 + cvector2;
   cout << "The result of vector addition:" << endl;
   cout << result;
   result = 10 * cvector2;
   cout << "The result of 10 * cvector2:" << endl;
   cout << result;
   return 0;
   }
```

Output

```
Here is cvector1:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(3.3, 102.8)
Here is cvector2:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(33.3, 81)
The result of vector addition:
```

```
[0]=(0, 0)   [1]=(2.2, 0)   [2]=(4.4, 0)   [3]=(36.6, 183.8)
The result of 10 * cvector2:
[0]=(0, 0)   [1]=(11, 0)   [2]=(22, 0)   [3]=(333, 810)
```

Overloading Operator Functions

See also Operators

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);   // same as c3 = c1 + c2
```

Apart from new and delete, which have their own rules, an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions =, ( ), [ ] and -> must be nonstatic member functions.

Enumerations can have overloaded operators. However, the operator functions =, ( ), [ ], and -> cannot be overloaded for enumerations.

Overloaded Operators and Inheritance

See also Operators

With the exception of the assignment function operator =( ), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.

Overloading the new and delete Operators

See also Operators

The operators new and delete can be overloaded to provide alternative free storage (heap) memory-management routines:

- A user-defined operator **new** must return a **void*** and must have a size_t as its first argument.
- A user-defined operator **delete** must have a **void** return type and **void*** as its first argument; a second argument of type size_t is optional.

Overloading Unary Operators

See also Operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

- Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator.
- With C++ 2.1, when an operator++ or operator- - is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator++ or operator- -. You can only overload a postfix operator++ or operator- - by defining it as a member function taking an int parameter or as a nonmember function taking one class and one int parameter.

When only the prefix version of an operator++ or operator- - is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in
  function func()
Warning: Overloaded prefix 'operator --' used as a postfix operator in
  function func()
```

Overloading Binary Operators

See also Operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Overloading the Assignment Operator =

See also Operators

The assignment operator=( ) can be overloaded by declaring a nonstatic member function. For example,

```
class String {
      .
      .
      .
   String& operator = (String& str);
      .
      .
      .
   String (String&);
   ~String();
}
```

This code, with suitable definitions of String::operator =(), allows string assignments str1 = str2 in the usual sense. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class X, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class X:

```
X& X::operator = (const X& source)
{
   // memberwise assignment
}
```

Overloading the Function Call Operator ( )

See also Operators

Syntax

```
postfix-expression ( <expression-list> )
```

Description

In its ordinary use as a function call, the postfix-expression must be a function name, or a pointer or reference to a function. When the postfix-expression is used to make a member function call, postfix-expression must be a class member function name or a pointer-to-member expression used to select a class member function. In either case, the postfix-expression is followed by the optional expression-list (possibly empty).

A call *X(arg1, arg2)*, where *X* is an object class *X*, is interpreted as *X*.**operator()**(*arg1, arg2*).

The function call operator, **operator()()**, can only be overloaded as a nonstatic member function.

Overloading the Subscript Operator [ ]

Syntax

```
postfix-expression [ expression ]
```

Description

The corresponding operator function is operator[]() this can be user-defined for a class *X* (and any derived classes). The expression *X*[*y*], where *X* is an object of class *X*, is interpreted as `x.operator[](y)`.

The **operator[]()** can only be overloaded as a nonstatic member function.

Overloading the Class Member Access Operator ->

See also Operators

Syntax

```
postfix-expression -> primary-expression
```

Description

The expression `x->m`, where *x* is a **class** *X* object, is interpreted as `(x.operator->())->m`, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

The **operator->()** can only be overloaded as a nonstatic member function.

const_cast

See also

Syntax

```
const_cast< T > (arg)
```

Description

Use the **const_cast** operator to add or remove the **const** or **volatile** modifier from a type.

In the statement, `const_cast< T > (arg)`, *T* and *arg* must be of the same type except for **const** and **volatile** modifiers. The cast is resolved at compile time. The result is of type *T*. Any number of const or volatile modifiers can be added or removed with a single **const_cast** expression.

A pointer to **const** can be converted to a pointer to non-**const** that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A **const** object or a reference to **const** cast results in a non-**const** object or reference that is otherwise an identical type.

The **const_cast** operator performs similar typecasts on the **volatile** modifier. A pointer to volatile object can be cast to a pointer to non-**volatile** object without otherwise changing the type of the object. The result is a pointer to the original object. A **volatile**-type object or a reference to **volatile**-type can be converted into an identical non-**volatile** type.

dynamic_cast

See also Example

In the expression, `dynamic_cast< T > (ptr)`, *T* must be a pointer or a reference to a defined class type or **void***. The argument *ptr* must be an expression that resolves to a pointer or reference.

If `T` is void* then *ptr* must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if *T* is a pointer and *ptr* is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If *T* is a reference and `ptr` is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, dynamic_cast< T > (ptr) converts *ptr* to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the Bad_cast exception is thrown.

Note: Runtime type identification (RTTI) is required for **dynamic_cast**.

// dynamic_cast Example

```
// HOW TO MAKE DYNAMIC CASTS
// This program must be compiled with the -RT (Generate RTTI) option.
#include <iostream.h>
#include <typeinfo.h>

class Base1
{
   // In order for the RTTI mechanism to function correctly,
   // a base class must be polymorphic.
   virtual void f(void) { /* A virtual function makes the class polymorphic
  */ }
};

class Base2 { };
class Derived : public Base1, public Base2 { };

int main(void) {
   try {
      Derived d, *pd;
      Base1 *b1 = &d;

      // Perform a downcast from a Base1 to a Derived.
      if ((pd = dynamic_cast<Derived *>(b1)) != 0) {
           cout << "The resulting pointer is of type "
                << typeid(pd).name() << endl;
         }
      else throw Bad_cast();

      // Attempt cast across the hierarchy.  That is, cast from
      // the first base to the most derived class and then back
      // to another accessible base.
      Base2 *b2;
      if ((b2 = dynamic_cast<Base2 *>(b1)) != 0) {
          cout << "The resulting pointer is of type "
               << typeid(b2).name() << endl;
         }
      else throw Bad_cast();
      }
   catch (Bad_cast) {
      cout << "dynamic_cast failed" << endl;
      return 1;
      }
   catch (...) {
      cout << "Exception handling error." << endl;
      return 1;
      }

   return 0;
}
```

reinterpret_cast

See also Example

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, `reinterpret_cast< T > (arg)`, `T` must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral `arg` can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

// reinterpret_cast Example

```
// Use reinterpret_cast<Type>(expr) to replace (Type)expr casts
// for conversions that are unsafe or implementation dependent.

void func(void *v) {
        // Cast from pointer type to integral type.
        int i = reinterpret_cast<int>(v);

  .
  .
  .
}

void main() {
   // Cast from an integral type to pointer type.
   func(reinterpret_cast<void *>(5));

   // Cast from a pointer to function of one type to
   // pointer to function of another type.
   typedef void (* PFV)();

   PFV pfunc = reinterpret_cast<PFV>(func);

   pfunc();
   }
```

Scope resolution operator ::

The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifiers by prefixing it with the resolution operator. To access a nested member name by specifying the class name and using the resolution operator. Therefore, `Alpha::func( )` and `Beta::func( )` are two different functions.

static_cast

See also

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, `static_cast< T > (arg)`, `T` must be a pointer, reference, arithmetic type, or enum type. The arg-type must match the T-type. Both `T` and `arg` must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using **static_cast** achieves exactly the same thing.

Integral types can be converted to enum types. A request to convert `arg` to a value that is not an element of **enum** is undefined.

The null pointer is converted to itself.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class X to a pointer to some class Y if X is a base class for Y. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from Y to X
- if X is not a virtual base class

An object can be explicitly converted to a reference type X& if a pointer to that object can be explicitly converted to an X*. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.
An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.
A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When `T` is a reference the result of `static_cast< T > (arg)` is an lvalue. The result of a pointer or reference cast refers to the original expression.

Predefined Macros

See also

Borland C++ predefines certain global identifiers known as manifest constants. Most global indentifers begin and end with two underscores (__).

Note: For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

See also the description of memory-model macros.

Macro	Value	What Macro Is/Does
_ _BCOPT_ _	1	Defined in any compiler that has an optimizer
_ _BCPLUSPLUS_ _	0x340	Defined if you've selected C++ compilation; will increase in later releases
_ _BORLANDC_ _	0x500	Version number
_ _CDECL_ _	1	Defined if Calling Convention is set to C; otherwise undefined
_CHAR_UNSIGNED	1	Defined by default indicating that the default **char** is **unsigned char**. Use the **-K** option to undefine this macro.
_ _CONSOLE_ _		Available only for the 32-bit compiler. When defined, the macro indicates that the program is a console application.
_CPPUNWIND	1	Enable stack unwinding.This is true by default; use **-xd-** to disable.
_ _cplusplus	1	Defined if in C++ mode; otherwise, undefined
_ _DATE_ _	String literal	Date when processing began on the current file
_ _DLL_ _	1	Defined if Prolog/Epilog Code Generation is set to Windows DLL; otherwise undefined
_ _FILE_ _	String literal	Name of the current file being processed
_ _LINE_ _	Decimal constant	Number of the current source file line being processed
_ _DLL_ _	1	True whenever **-WD** option is used.
_M_IX86	1	Always defined. The default value is 300. You can change the value to 400 or 500 by using the **/4** or **/5** options.
_ _MSDOS_ _	1	Integer constant
_ _MT_ _	1	Defined only for the 32-bit compiler if the `-WM` option is used. It specifies that the multithread library is to be linked. Multithread is on by default.
_ _OVERLAY_ _	1	Specific to the Borland C and C++ family of compilers. It is predefined as 1 if you compile a module with the -Y option (enable overlay support). If you do not enable overlay support, this macro is undefined.
_ _PASCAL_ _	1	Defined if Calling Convention is set to Pascal; otherwise undefined
_ _STDC_ _	1	Defined if you compile with the Keywords option set to ANSI; otherwise, undefined
_ _TCPLUSPLUS_ _	0x340	Version number
_ _TEMPLATES_ _	1	Specific to the Borland C++ compilers. It is defined as 1 for C++ files (meaning that Borland C++ supports templates); otherwise, it is undefined.

_ _TIME_ _	String literal	Time when processing began on the current file
_ _TLS_ _	1	Always true when the 32-bit compiler is used.
_ _TURBOC_ _	0x460	Will increase in later releases
_WCHAR_T	1	Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_WCHAR_T_DEFINED		1 Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_Windows		Defined for Windows 16- and 32-bit compilations
_ _WIN32_ _	1	Always defined for the 32-bit compiler. It is defined for console and GUI applications.

Note: _ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _STDC_ _, and _ _TIME_ _ **cannot** appear immediately following a #define or #undef directive.

€Header Files Summary

See also

Header files, also called include files, provide function prototype declarations for library functions. Data types and symbolic constants used with the library functions are also defined in them, along with global variables defined by Borland C++ and by the library functions. The Borland C++ library follows the ANSI C standard on names of header files and their contents.

Note: The middle column indicates C++ header files and header files defined by ANSI C.

alloc.h		Declares memory-management functions (allocation, deallocation, and so on).
assert.h	ANSI C	Defines the assert debugging macro.
bcd.h	C++	Declares the C++ class bcd and the overloaded operators for bcd and bcd math functions.
bios.h		Declares various functions used in calling IBM-PC ROM BIOS routines.
bwcc.h		Defines the Borland Windows Custom Control interface.
checks.h	C++	Defines the class diagnostic macros.
complex.h	C++	Declares the C++ complex math functions.
conio.h		Declares various functions used in calling the operating system console I/O routines.
constrea.h	C++	Defines the conbuf and constream classes.
cstring.h	C++	Defines the string classes.
ctype.h	ANSI C	Contains information used by the character classification and character conversion macros (such as isalpha and toascii).
date.h	C++	Defines the date class.
_defs.h		Defines the calling conventions for different application types and memory models.
dir.h		Contains structures, macros, and functions for working with directories and path names.
direct.h		Defines structures, macros, and functions for dealing with directories and path names.
dirent.h		Declares functions and structures for POSIX directory operations.
dos.h		Defines various constants and gives declarations needed for DOS and 8086-specific calls.
errno.h	ANSI C	Defines constant mnemonics for the error codes.
except.h	C++	Declares the exception-handling classes and functions.
excpt.h		Declares C structured exception support.
fcntl.h		Defines symbolic constants used in connection with the library routine open.
file.h	C++	Defines the file class.
float.h	ANSI C	Contains parameters for floating-point routines.
fstream.h	C++	Declares the C++ stream classes that support file input and output.
generic.h	C++	Contains macros for generic class declarations.

io.h		Contains structures and declarations for low-level input/output routines.
iomanip.h	C++	Declares the C++ streams I/O manipulators and contains templates for creating parameterized manipulators.
iostream.h	C++	Declares the basic C++ streams (I/O) routines.
limits.h	ANSI C	Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.
locale.h	ANSI C	Declares functions that provide country- and language-specific information.
malloc.h		Declares memory-management functions and variables.
math.h	ANSI C	Declares prototypes for the math functions and math error handlers.
mem.h		Declares the memory-manipulation functions. (Many of these are also defined in string.h.)
memory.h		Contains memory-manipulation functions.
new.h	C++	Access to _new_handler, and set_new_handler.
_nfile.h		Defines the maximum number of open files.
_null.h		Defines the value of NULL.
process.h		Contains structures and declarations for the spawn... and exec... functions.
search.h		Declares functions for searching and sorting.
setjmp.h	ANSI C	Declares the functions longjmp and setjmp and defines a type jmp_buf that these functions use.
share.h		Defines parameters used in functions that make use of file-sharing.
signal.h	ANSI C	Defines constants and declarations for use by the signal and raise functions.
stdarg.h	ANSI C	Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as vprintf, vscanf, and so on).
stddef.h	ANSI C	Defines several common data types and macros.
stdio.h	ANSI C	Defines types and macros needed for the standard I/O package defined in Kernighan and Ritchie and extended under UNIX System V. Defines the standard I/O predefined streams stdin, stdout, stdprn, and stderr and declares stream-level I/O routines.
stdiostr.h	C++	Declares the C++ (version 2.0) stream classes for use with stdio FILE structures. You should use iostream.h for new code.
stdlib.h	ANSI C	Declares several commonly used routines such as conversion routines and search/sort routines.
string.h	ANSI C	Declares several string-manipulation and memory-manipulation routines.
strstrea.h	C++	Declares the C++ stream classes for use with byte arrays in memory.
sys\locking.h		Contains definitions for mode parameter of locking

		function.
sys\stat.h		Defines symbolic constants used for opening and creating files.
sys\timeb.h		Declares the function ftime and the structure timeb that ftime returns.
sys\types.h		Declares the type time_t used with time functions.
thread.h	C++	Defines the thread classes.
time.h	ANSI C	Defines a structure filled in by the time-conversion routines asctime, localtime, and gmtime, and a type used by the routines ctime, difftime, gmtime, localtime, and stime. It also provides prototypes for these routines.
typeinfo.h	C++	Declares the run-time type information classes.
utime.h		Declares the utime function and the utimbuf struct that it returns.
values.h		Defines important constants, including machine dependencies; provided for UNIX System V compatibility.
varargs.h		Definitions for accessing parameters in functions that accept a variable number of arguments. Provided for UNIX compatibility; you should use stdarg.h for new code.

Using precompiled headers

See also Header Files

Borland C++ can generate (and subsequently use) precompiled headers to speed up your project compile times.

Precompiled headers are header files that are compiled once, then used over and over again in their compiled state.

You can use a precompiled header if a compilation uses one or more of the same header files, the same compiler options, the same macro defines, and so on, as is contained in the precompiled header file.

To control the use of precompiled headers, do one of the following:

- From within the IDE, turn on the Precompiled Headers option in the Compiler settings page of the Project Options dialog box. The IDE bases the name of the precompiled header file on the project name, creating<*PROJECT_NAME*>.CSM.
- From the command line, use the following command-line options: **-H=<filename>**, **-Hc**, **-H<filename>**, and **-Hu**. See Precompiled Headers (Project Options) for more information.
- From within your code, use the hdrfile and hdrstop pragmas.

Setting File Names

The compilers store all precompiled headers in one file, using the following naming conventions:

- The 16-bit command-line compiler names the precompiled header file BCDEF.CSM
- The 32-bit command-line compiler names the precompiled header file BC32DEF.CSM
- The IDE names the precompiled header file <*PROJECT_NAME*>.CSM

Note: To explicitly set the precompiled file name from the command line, use the `-H=<filename>` option or the `#pragma` **`hdrfile`** directive.

Precompiled header file overview

See also

When compiling C and C++ programs, the compiler can spend up to half its time parsing header files. When the compiler parses a header file, it enters declarations and definitions into its symbol table.

Precompiled headers cut this process short by creating and storing a binary image of the symbol table on disk. By directly loading a binary image of the symbol table, the compiler can increase the speed of this step by over ten times. The disadvantage is that precompiled header files can become quite large because they can contain the symbol table images for all the **#include** files encountered in your sources.

If, while compiling a source file, Borland C++ discovers that the first **#include** files are identical to those of a previous compilation (of either the same or different source), it loads the binary image for those **#include** files and parses the remaining **#include** files.

For a given module, either all or none of the precompiled headers are used--if compilation of any included header file fails, the precompiled header file isn't updated for that module.

Precompiled header limits

See also

When using precompiled headers, BCDEF.CSM can become very large because it contains symbol table images for all sets of includes encountered in your sources. If you don't have sufficient disk space, you'll get a warning saying the write failed because of the precompiled headers. To fix this, you must provide more disk space and retry the compile. For information on reducing the size of the BCDEF.CSM file, see Optimizing precompiled headers.

If you're using large macros in a makefile in addition to using precompiled headers, there is a limit on the macro size: 4K for 16-bit applications and 16K for 32-bit applications.

If a header file contains any code, it can't be precompiled. For example, although C++ class definitions can appear in header files, you should ensure that only inline member functions are defined in the header and heed warnings such as "`Functions containing reserved word are not expanded inline`."

Precompiled header rules

See also

The following rules apply when you create and use precompiled headers:

1) A header that contains code can't be precompiled. For example, although C++ class definitions can appear in header files, make sure that only inline member functions are defined in the header. Heed warnings such as "`Functions containing 'for' are not expanded inline`".

2) In order to use a previously generated precompiled header, the source file must:

- Have the same set of include files, in the same order, as the precompiled header
- Have the same macros defined with identical values as the precompiled header
- Use the same language (C or C++) as the precompiled header
- Use header files with identical time stamps as the precompiled header

3) In addition, the following option settings must be identical to those used when you generated the precompiled header:

- Memory model, including SS != DS (**-m*x***)
- Underscores on externs (**-u**)
- Maximum identifier length (**-iL**)
- Target DOS or Windows (**-W** or **-W*x***)
- Generate word alignment (**-a**)
- Pascal calls (**-p**)
- Treat enums as integers (**-b**)
- Default char is unsigned (**-K**)
- Virtual table control (**-V*x*** and **-Vm*x***)
- Expand intrinsic functions inline (**-Oi**)
- Templates (**-J*x***)
- String literals in code segment (**-dc**, 16-bit only)
- Debugging information (**-v**, **-vi**, and **-R**)
- Far variables (**-F*x***)
- Language compilance (**-A**)
- C++ compile (**-P**)
- DOS overlay-compatible code (**-Y**)

4) If you’re using large macros in addition to using precompiled headers, the compiler limits the size of the macros as following:

- 4K macros for 16-bit applications
- 16K macros for 32-bit applications

Optimizing precompiled headers

See also

For the most efficiently compiled precompiled headers, follow these rules:

- Arrange your header files in the same sequence in all source files.
- Put the largest header files first.
- Prime the precompiled header file with often-used initial sequences of header files.
- Use `#pragma` **`hdrstop`** to terminate the list of header files at well-chosen places. This lets you make the list of header files in different sources look similar to the compiler.

For example, suppose you have the following two source files (A_SOURCE.CPP and B_SOURCE.CPP),which both include windows.h and myhdr.h:

```
/* A_SOURCE.CPP */
#include <windows.h>
#include "myhdr.h"
#include "xxx.h"
// ...

/* B_SOURCE.CPP */
#include "yyy.h
#include <string.h>
#include "myhdr.h"
#include <windows.h>
// ...
```

To optimize the precompiled headers for these source files, you would rearrange the beginning of B_SOURCE.CPP as follows:

```
/* Revised B_SOURCE.CPP */
#include <windows.h>
#include "myhdr.h"
#include "yyy.h"
#include <string.h>
// ...
```

Now, windows.h and myhdr.h are in the same order in both A_SOURCE.CPP and B_SOURCE.CPP, and they are both located at the beginning of the `#include` list.

In addition, you could also create a new source file called PREFIX.CPP which contains only the matching header files, like this:

```
/* PREFIX.CPP */
#include <windows.h>
#include "myhdr.h"
```

If you compile PREFIX.CPP first (or insert a `#pragma` **`hdrstop`** in both A_SOURCE.CPP and B_SOURCE.CPP), the net effect is that after the initial compilation of PREFIX.CPP, both A_SOURCE.CPP and B_SOURCE.CPP will be able to load the symbol table produced by PREFIX.CPP. The copmiler will then need to parse only xxx.h for A_SOURCE.CPP, and yyy.h and strings.h for B_SOURCE.CPP.

alloc.h

See also Header Files

Declares memory-management functions (allocation, deallocation, and so on).

Functions

calloc
farcalloc
farfree
farmalloc
farrealloc
free
heapcheck
heapcheckfree
heapchecknode
heapfillfree
heapwalk
malloc
realloc

Constants, Data Types and Global Variables

NULL
ptrdiff_t
size_t

assert.h

See also Header Files

Defines the assert debugging macro.

Functions

assert

bios.h

See also Header Files

Declares various functions used in calling IBM-PC ROM BIOS routines.

Functions

_bios_equip
_bios_disk
_bios_equiplist
_bios_keybrd
_bios_memsize
_bios_serialcom
_bios_timeofday
bioscom
biosequip
bioskey
biosmemory
biosprint
biostime

conio.h

See also Header Files

Declares various functions used in calling the operating system console I/O routines.

Functions

cgets
clreol
clrscr
cprintf
cputs
cscanf
delline
getch
getche
getpass
gettext
gettextinfo
gotoxy
highvideo
inp
inport
inportb
inpw
insline
kbhit
lowvideo
movetext
normvideo
outp
outport
outportb
outpw
putch
puttext
_setcursortype
textattr
textbackground
textcolor
textmode
ungetch
wherex
wherey
window

ctype.h

See also Header Files

Contains information used by the character classification and character conversion macros.

Functions and Macros

isalnum
isalpha
isascii
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
toascii
_tolower
tolower
_toupper
toupper

Constants, Data Types and Global Variables

_IS_CTL
_IS_DIG
_IS_HEX
_IS_LOW
_IS_PUN
_IS_SP
_IS_UPP

dir.h

See also Header Files

Contains structures, macros, and functions for working with directories and path names.

Functions

chdir
findfirst
findnext
fnmerge
fnsplit
getcurdir
getcwd
getdisk
mkdir
mktemp
rmdir
searchpath
setdisk

Constants, Data Types and Global Variables

DIRECTORY
DRIVE
EXTENSION
ffblk
FILENAME
MAXDIR
MAXDRIVE
MAXEXT
MAXFILE
MAXPATH

direct.h

See also Header Files

Defines structures, macros, and functions for dealing with directories and path names.

Includes

DIR.H

Functions

_chdrive

_getdcwd

dirent.h

See also Header Files

Declares functions and structures for POSIX directory operations.

Functions

closedir

opendir

readdir

rewinddir

dos.h

See also Header Files

Defines various constants and gives declarations needed for DOS and 8086-specific calls.

Functions

allocmem (in Borland C++ DOS Support Help)
bdos
bdosptr
_chain_intr
_chmod
country
ctrlbrk
delay (in Borland C++ DOS Support Help)
disable
_dos_allocmem (in Borland C++ DOS Support Help)
_dos_close
_dos_commit
_dos_creat
_dos_creatnew
dosexterr
_dos_findfirst
_dos_findnext
_dos_freemem (in Borland C++ DOS Support Help)
_dos_getdate
_dos_getdiskfree
_dos_getdrive
_dos_getfileattr
_dos_getftime
_dos_gettime
_dos_getvect
_dos_keep (in Borland C++ DOS Support Help)
_dos_open
_dos_read
_dos_setblock (in Borland C++ DOS Support Help)
_dos_setdate
_dos_setdrive
_dos_setfileattr
_dos_settime
_dos_setvect
dostounix
_dos_write
emit
enable
FP_OFF
FP_SEG
geninterrupt

getcbrk
getdate
getdfree
getdta
getfat
getfatd
getftime
getpsp
gettime
getvect
getverify
_harderr (in Borland C++ DOS Support Help)
_hardresume (in Borland C++ DOS Support Help)
_hardretn (in Borland C++ DOS Support Help)
inport
inportb
int86
int86x
intdos
intdosx
intr
keep (in Borland C++ DOS Support Help)
MK_FP
nosound (in Borland C++ DOS Support Help)
outport
outportb
parsfnm
peek
peekb
poke
pokeb
randbrd (in Borland C++ DOS Support Help)
randbwr (in Borland C++ DOS Support Help)
segread
setcbrk
setdate
setdta
settime
setvect
setverify
sleep
sound (in Borland C++ DOS Support Help)
unixtodos
unlink

Constants, Data Types and Global Variables

_8087
_argc
_argv
COUNTRY
date
devhdr
dfree
diskfree_t
dosdate_t
DOSERROR
dostime_t
_doserrno
dosSearchInfo
errno
_environ
fatinfo
fcb
FA_*
ffblk
_heaplen (in Borland C++ DOS Support Help)
NFDS
_osmajor
_osminor
_osversion
_ovrbuffer (in Borland C++ DOS Support Help)
_psp
REGPACK
REGS
SEEK_CUR
SEEK_END
SEEK_SET
SREGS
_stklen (in Borland C++ DOS Support Help)
time
_version
xfcb

errno.h

See also Header Files

Defines constant mnemonics for the error codes.

Constants, Data Types and Global Variables

_doserrno

errno

_sys_errlist

_sys_nerr

error number definitions

fcntl.h

See also Header Files

Defines open flags for open and similar library functions.

Functions

_fmode

_pipe

Constants

O_APPEND

O_BINARY

O_CHANGED

O_CREAT

O_DENYALL

O_DENYNONE

O_DENYREAD

O_DENYWRITE

O_DEVICE

O_EXCL

O_NOINHERIT

O_RDONLY

O_RDWR

O_TEXT

O_TRUNC

O_WRONLY

float.h

See also Header Files

Contains parameters for floating-point routines.

Functions

_clear87

_fpreset

_status87

Constants, Data Types and Global Variables

CW_DEFAULT

FPE_EXPLICITGEN

FPE_INEXACT

FPE_INTDIV0

FPE_INTOVFLOW

FPE_INVALID

FPE_OVERFLOW

FPE_UNDERFLOW

FPE_ZERODIVIDE

ILL_EXECUTION

ILL_EXPLICITGEN

SEGV_BOUND

SEGV_EXPLICITGEN

generic.h

See also Header Files

Contains macros for generic class declarations.

io.h

See also Header Files

Contains structures and declarations for low-level input/output routines

Functions

access
chmod
chsize
close
creat
creatnew
creattemp
dup
dup2
eof
filelength
_get_osfhandle
getftime
_InitEasyWin
ioctl
isatty
lock
locking
lseek
mktemp
open
_open_osfhandle
_pipe
read
remove
rename
_rtl_chmod
_rtl_close
_rtl_creat
_rtl_open
_rtl_read
_rtl_write
setftime
setmode
sopen
tell
umask
unlink
unlock
write

Constants, Data Types and Global Variables

ftime structure

HANDLE_MAX

fseek/lseek modes

iomanip.h

See also Header Files

Declares the C++ streams I/O manipulators and contains macros for creating parameterized manipulators.

Includes

iostream.h

Classes

iapply

imanip

ioapp

iomanip

oapp

omanip

sapp

smanip

Overloaded Operators

<< >>

limits.h

See also Header Files

Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.

Constants, Data Types and Global Variables

CHAR_BIT
CHAR_MAX
CHAR_MIN
INT_MAX
INT_MIN
LONG_MAX
LONG_MIN
SCHAR_MAX
SCHAR_MIN
SHRT_MAX
SHRT_MIN
UCHAR_MAX
UINT_MAX
ULONG_MAX
USHRT_MAX

locale.h

See also Header Files

Declares functions that provide information specific to languages and countries.

Functions

localeconv

setlocale

Constants, Data Types and Global Variables

LC_ALL

LC_COLLATE

LC_CTYPE

LC_MONETARY

LC_NUMERIC

LC_TIME

lconv (struct)

NULL

malloc.h

See also Header Files

Declares memory-management functions and variables.

Includes

ALLOC.H

Functions

_heapchk
_heapmin
_heapset
_msize
_rtl_heapwalk
stackavail

math.h

See also Header Files

Declares prototypes for the math functions and math error handlers.

Functions

abs
acos, acosl
asin, asinl
atan, atanl
atan2, atan2l
atof, _atold
cabs, cabsl
ceil, ceill
cos, cosl
cosh, coshl
exp, expl
fabs, fabs
floor, floorl
fmod, fmodl
frexp, frexpl
hypot, hypotl
labs
ldexp, ldexpl
log, logl
log10, log10l
_matherr, _matherrl
modf, modfl
poly, polyl
pow, powl
pow10, pow10l
sin, sinl
sinh, sinhl
sqrt, sqrtl
tan, tanl
tanh, tanhl

Constants, Data Types and Global Variables

complex (struct)
_complexl (struct)
EDOM
ERANGE
exception (struct)
_exceptionl (struct)
HUGE_VAL
M_E
M_LOG2E
M_LOG10E

M_LN2
M_LN10
M_PI
M_PI_2
M_PI_4
M_1_PI
M_2_PI
M_1_SQRTPI
M_2_SQRTPI
M_SQRT2
M_SQRT_2
_mexcep

mem.h

See also Header Files

Declares the memory-manipulation functions. (Many of these are also defined in string.h.)

Functions

_fmemccpy
_fmemchr
_fmemcmp
_fmemcpy
_fmemicmp
_fmemmove
_fmemset
_fmovmem
memccpy
memchr
memcmp
memcpy
memicmp
memmove
memset
movedata
movmem
setmem

Constants, Data Types and Global Variables

NULL
ptrdiff_t
size_t

memory.h

See also Header Files

Contains memory-manipulation functions.

Includes

MEM.H

new.h

See also Header Files

Provides access to the the following functions:

set_new_handler

_new_handler (global variable)

process.h

See also Header Files

Contains structures and declarations for the spawn... and exec... functions.

Functions

abort
_beginthread
_beginthreadNT
_c_exit
_cexit
cwait
_endthread
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe
exit
_exit
getpid
spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe
wait

Constants, Data Types and Global Variables

P_DETACH
P_NOWAIT
P_NOWAITO
P_OVERLAY
P_WAIT

search.h

See also Header Files

Declares functions for searching and sorting.

Functions

bsearch

lfind

lsearch

qsort

setjmp.h

Declares the functions longjmp and setjmp and defines a type jmp_bufj that these functions use.

Functions

longjmp

setjmp

Constants, Data Types and Global Variables

jmp_buf

share.h

See also Header Files

Defines parameters used in functions that make use of file-sharing.

Constants, Data Types and Global Variables

SH_COMPAT

SH_DENYNO

SH_DENYNONE

SH_DENYRD

SH_DENYRW

SH_DENYWR

signal.h

See also Header Files

Defines constants and declarations for use by the signal and raise functions.

Functions

raise

signal

Constants, Data Types and Global Variables

predefined signal handlers

sig_atomic_t type

SIG_DFL

SIG_ERR

SIG_IGN

SIGABRT

SIGFPE

SIGILL

SIGINT

SIGSEGV

SIGTERM

stdarg.h

See also Header Files

Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as vprintf, vscanf, and so on).

Macros

va_arg

va_end

va_start

Constants, Data Types and Global Variables

va_list

stddef.h

See also Header Files

Defines several common data types and macros.

Functions

offsetof

Constants, Data Types and Global Variables

NULL

ptrdiff_t

size_t

_threadid

wchar_t

stdio.h

See also Header Files

Defines types and macros needed for the standard I/O package defined in Kernighan and Ritchie and extended under UNIX System V. It defines the standard I/O predefined streams stdin, stdout, stdprn, and stderr, and declares stream-level I/O routines.

Functions

clearerr
fclose
fcloseall
fdopen
feof
ferror
fflush
fgetc
fgetchar
fgetpos
fgets
fileno
flushall
fopen
fprintf
fputc
fputchar
fputs
fread
freopen
fscanf
fseek
fsetpos
_fsopen
_fstrncpy
ftell
fwrite
getc
getchar
gets
getw
_pclose
perror
_popen
printf
putc
putchar
puts
putw
remove
rename
rewind
rmtmp
scanf
setbuf
setvbuf
spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe
sprintf
sscanf
strerror
_strerror
strncpy
tempnam
tmpfile
tmpnam
ungetc
unlink
vfprintf
vfscanf
vprintf
vscanf
vsprintf
vsscanf

Constants, Data Types and Global Variables

buffering modes
BUFSIZ
EOF
_F_BIN
_F_BUF
_F_EOF
_F_ERR
_F_IN
_F_LBUF
_F_OUT
_F_TERM
_F_WRIT
FILE
fpos_t
fseek/lseek modes
_IOFBF
_IOLBF
_IONBF
L_ctermid
L_tmpnam
SEEK_CUR
SEEK_END
SEEK_SET
size_t
stdaux
stderr
stdin
stdout
stdprn
SYS_OPEN

_F_RDWR NULL TMP_MAX
_F_READ FOPEN_MAX

stdiostr.h

See also Header Files

Declares the C++ (version 2.0) stream classes for use with stdio FILE structures. You should use iostream.h for new code.

Includes

IOSTREAM.H

STDIO.H

stdlib.h

See also Header Files

Declares several commonly used routines such as conversion routines and search/sort routines.

Functions

abort	labs	realloc
abs	ldiv	_rotl
atexit	lfind	_rotr
atof	_lrotl	_searchenv
atoi	_lrotr	_searchstr
atol	lsearch	_splitpath
bsearch	ltoa	srand
calloc	_makepath	strtod
_crotr	malloc	strtol
div	max	_strtold
ecvt	mblen	strtoul
exit	mbstowcs	swab
_exit	mbtowc	system
fcvt	min	time
free	putenv	ultoa
_fullpath	qsort	wcstombs
gcvt	rand	wctomb
getenv	random	
itoa	randomize	

Constants, Data Types and Global Variables

div_t
_doserrno
environ
errno
EXIT_FAILURE
EXIT_SUCCESS
_fmode
ldiv_t
NULL
_osmajor
_osminor
RAND_MAX
size_t
sys_errlist
sys_nerr
_version
wchar_t

string.h

See also Header Files

Declares several string-manipulation and memory-manipulation routines.

Includes

LOCALE.H

Functions

_fmemccpy	_fstrset	strdup
_fmemchr	_fstrspn	strdup
_fmemcmp	_fstrstr	strerror
_fmemcpy	_fstrtok	_strerror
_fmemicmp	_fstrupr	stricmp
_fmemset	memccpy	strlen
_fstr*	memchr	strlwr
_fstrcat	memcmp	strncat
_fstrchr	memcpy	strncmp
_fstrcmp	memicmp	strncmpi
_fstrcpy	memmove	strncpy
_fstrcspn	memset	strnicmp
_fstrdup	movedata	strnset
_fstricmp	movmem	strpbrk
_fstrlen	setmem	strrchr
_fstrlwr	stpcpy	strrev
_fstrncat	strcat	strset
_fstrncmp	strchr	strspn
_fstrncpy	strcmp	strstr
_fstrnicmp	strcmp	strtok
_fstrnset	strcmpi	strupr
_fstrpbrk	strcoll	strxfrm
_fstrrchr	strcpy	
_fstrrev	strcspn	

Constants, Data Types and Global Variables

size_t

sys\locking.h

See also Header Files

Contains definitions for mode parameter of locking function.

Constants

LK_LOCK

LK_NBLCK

LK_NBRLCK

LK_RLCK

LK_UNLCK

sys\stat.h

See also Header Files

Defines symbolic constants used for opening and creating files.

Includes

SYS\TYPES.H

Functions

chmod

fstat

stat

Constants, Data Types and Global Variables

file status bits

stat structure

sys\timeb.h

See also Header Files

Functions

ftime

Constants, Data Types and Global Variables

timeb structure

_timezone

sys\types.h

See also Header Files

Constants, Data Types and Global Variables

time_t

time.h

See also Header Files

Defines a structure filled in by time-conversion routines asctime, localtime, and gmtime, and a type used by the routines ctime, difftime, gmtime, localtime and stime. It also provides prototypes for these routines.

Functions

asctime
clock
ctime
difftime
gmtime
localtime
mktime
randomize
stime
_strdate
strftime
_strtime
time
tzset

Constants, Data Types and Global Variables

CLK_TCK
clock_t
daylight
size_t
time_t
timezone
tm
tzname

Classes

Time classes

utime.h

See also Header Files

Declares the utime function and the utimbuf struct that it returns.

Function

utime

Constants, Data Types and Global Variables

time_t

utimbuf

values.h

See also Header Files

Defines UNIX compatible constants for limits to float and double values.

BITSPERBYTE
DMAXEXP
DMAXPOWTWO
DMINEXP
DSIGNIF
FMAXEXP
FMAXPOWTWO
FMINEXP
FSIGNIF
_FEXPLEN
HIBITI
HIBITL
HIBITS
_LENBASE
MAXDOUBLE
MAXFLOAT
MAXINT
MAXLONG
MAXSHORT
MINDOUBLE
MINFLOAT

varargs.h

See also Header Files

Definitions for accessing parameters in functions that accept a variable number of arguments.

These macros are compatible with UNIX System V.

Use STDARG.H for ANSI C compatibility.

Note: You can't include both STDARG.H and VARARGS.H

Macros

va_start

va_arg

va_end

Type

va_list

excpt.h

See also Header Files

The excpt.h header file contains the declarations and prototypes for structured exception-handling values, types, and routines. Consult the Windows API documentation for more details.

bwcc.h

See also Header Files

The bwcc.h header file defines the interface for Borland Windows Custom Control library (BWCC).

For details on using the Borland Windows Custom Control library, see the Borland Windows Custom Controls Reference.

_defs.h

See also Header Files

The _defs.h header file contains common definitions for pointer size and calling conventions.

Calling Conventions

_RTLENTRY	Specifies the calling convention used by the Standard Run-time Library.
_USERENTRY	Specifies the calling convention the Standard Run-time Library expects user-compiled functions to use for callbacks.

Export (and Size for DOS) Information

_EXPCLASS	Exports the class if you are building a DLL version of a library.
_EXPDATA	Exports the data if you are building a DLL version of a library.
_EXPFUNC	Exports the function if you are building a DLL version of a library.

Note: These export macros are provided as examples only and should not be used to create user-defined functions.

_nfile.h

See also Header Files

The _nfile.h header file defines _NFILE_, which specifies the maximum number of open files you can have.

NFILE is defined as 50 for all applications.

_null.h

See also Header Files

The _null.h defines the value of NULL for different memory models and applications types:

Model	Value	
Flat	((void *)0)	if not C++ or Windows application
Flat	0	
Tiny	0	
Small	0	
Medium	0	
Large	0L	

Using Templates

See also

Templates, also called *generics* or *parameterized* types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates:

Exporting and importing templates

Template Syntax

Template Body Parsing

Function Templates

Class Templates

Implicit and Explicit Template Functions

Template Compiler Switches

Note: For some complete examples of templates and template-driven classes, see the source files for the ObjectWindows classes in the SOURCE\OWL directories.

Template body parsing

See also

Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

```
template <class T> class X : T
{
  Int  j;  // Error: Type name expected in template X<T>
};
```

Let's assume that *Int* hasn't been defined so far. This means that *Int* must be a member of the template argument *T*. But it also might just be a typing error and should be **int** instead of *Int*. Because the compiler can't guess the right meaning it issues an error message.

If you want to access types defined by a template argument you should use a **typedef** to make your intention clear to the compiler:

```
template <class T> class X : T
{
  typedef  T::Int  Int;
  Int  j;
};
```

You cannot just write

```
   typedef   T::Int;
```

as in earlier versions of the compiler. Not giving the **typedef** name was acceptable, but this now causes an error message.

All other templates mentioned inside the template body are declared or defined at that point. Therefore, the following example is ill-formed and will not compile:

```
template <class T> class  X
{
  void f(NotYetDefindedTemplate<T> x);
};
```

All template definitions must end with a semicolon. Earlier versions of the compiler did not complain if the semicolon was missing.

Function Templates

See also Using Templates

Consider a function *max(x, y)* that returns the larger of its two arguments. *x* and *y* can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters *x* and *y* to be declared at compile time. Without using templates, many overloaded versions of *max* are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the **#define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of *max(x, y)* is to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be. By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

```
template <class T> T max(T x, T y){
   return (x > y) ? x : y;
   };
```

The data type is represented by the template argument **<class T>**. When used in an application, the compiler generates the appropriate code for the *max* function according to the data type actually used in the call:

```
int i;
Myclass a, b;

int j = max(i,0);       // arguments are integers
Myclass m = max(a,b);   // arguments are type Myclass
```

Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator>()**, so you can use *max* with arguments of any type for which **operator>()** is defined.

Overriding a Template Function

Using Templates

The previous example is called a *function template* (or *generic function*, if you like). A specific instantiation of a function template is called a *template function*. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>

char *max(char *x, char *y){
   return(strcmp(x,y) > 0) ? x : y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit and Explicit Template Functions

Using Templates

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b){
        return  (a > b) ? a : b;
};

void    f(int i, char c){
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // no match for max(int,char)
        max(c, i);              // no match for max(char,int)
}
```

This code results in the following error messages:

Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)

If the user explicitly declares a template function, this function, on the other hand, will participate fully in overload resolution. See the example of explicit template function.

When searching for an exact match for template function parameters, trivial conversions are considered to be exact matches. See the example on trivial conversions.

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. See the example of base class referencing.

Example of base class referencing

```
template <class T> class B
{
  // class declarations
};
template <class T> class D : public B<T>
{
   // class declarations
};

template <class T> void func(B <T> *b)
{
   // function body
}
// This is illegal under ANSI C++: unresolved func( int )
// However, Borland C++ calls func( B<int> * ).
func( new D<int> );
```

Example of trivial conversions

```
template <class T> void func(const T)
{
    .
    .
    .
};
func(0);  // This is illegal under ANSI C++: unresolved func(int).
// However, Borland C++  allows func(const int) to be called.
```

Example of explicit template function

```
template<class T> T max(T a, T b) {
        return  (a > b) ? a : b;
};

// Declare explicit template function
int     max(int,int);

void    f(int i, char c)
{
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // calls max(int,int)
        max(c, i);              // calls max(int,int)
}
```

Class Templates

See also Using Templates Example

A class template (also called a *generic class* or *class generator*) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type parameter to the class, the system will generate type-safe class definitions on the fly.

As with function templates, an explicit *template class* definition can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol *Vector* must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

Class template definition

```
// An example for defining a template class.
template <class T> class Vector
{
   T *data;
   int size;
public:
   Vector(int);
   ~Vector( ) { delete[ ] data; }
   T& operator[ ] (int i) { return data[i]; }
};
// Note the syntax for out-of-line definitions.
template <class T> Vector<T>::Vector(int n)
{
   data = new T[n];
   size = n;
};

int main()
{
   Vector<int> x(5);   // Generate a vector to store five integers
   for (int i = 0; i < 5; ++i)
      x[i] = i;              //  Initialize the vector.
   return o;
}
```

Template Arguments

Using Templates

Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Non-type template arguments such as *size* can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;

Buffer<int, 2*N> b1;// OK
Buffer<float, i> b2;// Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Using Angle Brackets in Templates

Using Templates

Be careful when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between *x* and 100 would prematurely close the template argument list.

Using Type-safe Generic Lists in Templates

Using Templates

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
 public:
   void insert( void * );
   void *peek();
    .
    .
    .
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public Glist {
 public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo *)GList::peek(); }
      .
      .
      .
};
```

This is type-safe. *insert* will only take arguments of type pointer-to-*Foo* or object-derived-from-*Foo*, so the underlying container will only hold pointers that in fact point to something of type *Foo*. This means that the cast in *FooList::peek()* is always safe, and you've created a true *FooList*. Now, to do the same thing for a *BarList*, a *BazList*, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and be type-safe, you can once again use templates. See the example for type-safe generic list class.

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no run-time overhead imposed by this type safety.

Type-safe generic list class definition

```
template <class T> class List : public GList
{
public:
   void insert( T *t ) { GList::insert( t ); }
   T *peek() { return (T *)GList::peek(); }
    .
    .
    .
};

  // Create a List object of Foo types and name it fList.
  List<Foo> fList;

// Create a List object of Bar types and name it bList.
  List<Bar> bList;

// Create a List object of Baz types and name it zList.
  List<Baz> zList;
```

Eliminating Pointers in Templates

Using Templates

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is beneficial if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

```
template <class T> aBase {
    .
    .
    .
 private:
  T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf> {
    .
    .
    .
};
```

All the functions in *aBase* can call functions defined in *aFilebuf* directly, without having to go through a pointer. And if any of the functions in *aFilebuf* can be inlined, you'll get a speed improvement, because templates allow them to be inlined.

Template Compiler Switches

Using Templates

The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler.

For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

When using the **-Jg** family of switches, there are two basic approaches for generating template instances:

- Include the function body (for a function template) or member function and static data member definitions (for a template class) in the header file that defines the particular template, and use the default setting of the template switch (**-Jg**). If some instances of the template are user-defined, the declarations (prototypes, for example) for them should be included in the same header but preceded by **#pragma option -Jgx**. See the example for template header files.
- Compile all of the source files comprising the program with the **-Jgx** switch (causing external references to templates to be generated). In order to provide the definitions for all of the template instances, add a file (or files) to the program that includes the template bodies (including any user-defined instance definitions), and list all the template instances needed in the rest of the program to provide the necessary public symbol definitions. Compile the file (or files) with the **-Jgd** switch. See the example for separate file template compilation.

Separate file template compilation

```
// In vector.h
template <class elem, int size> class vector
{
   elem * value;
public:
   vector();
   elem & operator [ ] (int index) {
      return value[index];
      }
};
// In main.cpp source file.
#include "vector.h"
/** Let the compiler know that the following template instances will be
  defined elsewhere. **/
#pragma option -Jgx
// Use two instances of the vector template class.
vector<int, 100> int_100;
vector<char, 10> char_10;
int main( )
{
   return int_100[ 0 ] + char_10[ 0 ];
}

// In template.cpp source file.
#include <string.h>
#include "vector.h"
// Define any template bodies.
template <class elem, int size> vector <elem, size> :: vector()
{
   value = new elem[size];
   memset(value, 0, size * sizeof(elem) );
}
// Generate the necessary instances.
#pragma option -Jgd
typedef vector<int, 100> fake_int_100;
typedef vector<char, 10> fake_char_10;
```

Template header file

```
// Declare a template function and define it's body.
/* When this header file is included in a C++ source file, the sort template
  can be used without worrying about how the various instances are generated
  (with the exception of sort for integer arrays which is a user-defined
  instance. Its definition must be provided by the user. */
template<class T> void sort (T* array, int size)
{
   // Body of template goes here.
}
// Sorting of integer elements done by user-define instance.
#pragma option -Jgx
extern void sort(int *array, int size);
// Restore the template switch to its original state.
#pragma option -Jg
```

The main() Function

See also

Every C and C++ program must have a program-startup function.

- Console-based programs call the *main* function at startup.
- Windows GUI programs call the WinMain function at startup.

Where you place the startup function is a matter of preference. Some programmers place *main* at the beginning of the file, others at the end. Regardless of its location, the following points about *main* always apply.

- Arguments to main
- Wildcard Arguments
- Using -p (Pascal Calling Conventions)
- Value main( ) Returns

Arguments to main ()

The main() Function Example

Three parameters (arguments) are passed to *main* by the Borland C++ startup routine: *argc, argv*, and *env*.

- *argc*, an integer, is the number of command-line arguments passed to *main*, including the name of the executable itself.
- *argv* is an array of pointers to strings (**char** *[]).
  - *argv*[0] is the full path name of the program being run.
  - *argv*[1] points to the first string typed on the operating system command line after the program name.
  - *argv*[2] points to the second string typed after the program name.
  - *argv*[*argc*-1] points to the last argument passed to *main*.
  - *argv*[*argc*] contains NULL.
- *env* is also an array of pointers to strings. Each element of *env*[] holds a string of the form ENVVAR=value.
  - ENVVAR is the name of an environment variable, such as PATH or COMSPEC.
  - *value* is the value to which ENVVAR is set, such as C:\APPS;C:\TOOLS; (for PATH) or C:\DOS\COMMAND.COM (for COMSPEC).

If you declare any of these parameters, you *must* declare them exactly in the order given: *argc, argv, env*. For example, the following are all valid declarations of arguments to *main*:

```
int main()
int main(int argc)                  /* legal but very unlikely */
int main(int argc, char * argv[])
int main(int argc, char * argv[], char * env[])]
```

The declaration int main(int argc) is legal, but it is very unlikely that you would use *argc* in your program without also using the elements of *argv*.

The argument *env* is also available through the global variable _environ.

For all platforms, *argc* and *argv* are also available via the global variables _argc and _argv.

Example of how Arguments are Passed to main( )

Here is an example that demonstrates a simple way of using these arguments passed to main:

```
/* Program ARGS.C */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[]) {
   int i;

   printf("The value of argc is %d \n\n", argc);
   printf("These are the %d command-line arguments passed to"
          " main:\n\n", argc);

   for (i = 0; i < argc; i++)
      printf("   argv[%d]: %s\n", i, argv[i]);

   printf("\nThe environment string(s) on this system are:\n\n");

   for (i = 0; env[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, env[i]);
   return 0;
   }
```

Suppose you run ARGS.EXE at the command prompt with the following command line:

```
   C:> args first_arg "arg with blanks" 3  4 "last but one" stop!
```

Notice that you can pass arguments with embedded blanks by surrounding them with quotes, as shown by "argument with blanks" and "last but one" in this example command line.

The output of ARGS.EXE (assuming that the environment variables are set as shown here) would then be like this:

```
The value of argc is 7

These are the 7 command-line arguments passed to main:

   argv[0]: C:\BC5\ARGS.EXE
   argv[1]: first_arg
   argv[2]: arg with blanks
   argv[3]: 3
   argv[4]: 4
   argv[5]: last but one
   argv[6]: stop!

The environment string(s) on this system are

   env[0]: COMSPEC=C:\COMMAND.COM
   env[1]: PROMPT=$p $g
   env[2]: PATH=C:\SPRINT;C:\DOS;C:\BC5
```

The maximum combined length of the command-line arguments passed to *main* (including the space between adjacent arguments and the program name itself) is

- 128 for DOS
- 260 for Win16
- 255 for Win32

Wildcard Arguments

The main() Function Example

Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARGS.OBJ object file, which is included with Borland C++.

Note: Wildcard arguments are used only in console-mode applications.

Once WILDARGS.OBJ is linked into your program code, you can send wildcard arguments (such as *.*) to your *main* function. The argument will be expanded (in the *argv* array) to all files matching the wildcard mask. The maximum size of the *argv* array varies, depending on the amount of memory available in your heap.

If no matching files are found, the argument is passed unchanged. (That is, a string consisting of the wildcard mask is passed to *main*.)

Arguments enclosed in quotes ("...") are not expanded.

Example of using Wildcard Arguments with main( )

The following commands compile the file ARGS.C and link it with the wildcard expansion module WILDARGS.OBJ, then run the resulting executable file ARGS.EXE:

```
BCC  ARGS.C  WILDARGS.OBJ
ARGS  C:\BC5\INCLUDE\*.H  "*.C"
```

When you run ARGS.EXE, the first argument is expanded to the names of all the *.H files in your Borland C++ INCLUDE directory. Note that the expanded argument strings include the entire path. The argument *.C is not expanded because it is enclosed in quotes.

In the IDE, simply specify a project file from the Project menu) that contains the following lines:

```
ARGS
WILDARGS.OBJ
```

If you prefer the wildcard expansion to be the default, modify your standard CW32?.LIB library files to have WILDARGS.OBJ linked automatically. To do so, remove SETARGV and INITARGS from the libraries and add WILDARGS. The following commands invoke the Turbo librarian (TLIB) to modify all the standard library files (assuming the current directory contains the standard C and C++ libraries and WILDARGS.OBJ):

Window Users

```
tlib  CW32          -setargv  +wildargs
tlib  CW32MT        -setargv  +wildargs
tlib  -setargv      +wildargs
```

DOS Users

```
tlib  cs  -setargv  +wildargs
tlib  cc  -setargv  +wildargs
tlib  cm  -setargv  +wildargs
tlib  cl  -setargv  +wildargs
tlib  ch  -setargv  +wildargs
```

Using --p (Pascal Calling Conventions)

The main() Function

If you compile your program using Pascal calling conventions, you must remember to explicitly declare *main* as a C type. Do this with the __cdeclkeyword, like this:

```
int  _ _cdecl main(int argc, char* argv[], char* envp[])
```

The Value main() Returns

The main() Function

The value returned by *main* is the status code of the program: an **int**. If, however, your program uses the routine exit (or _exit) to terminate, the value returned by *main* is the argument passed to the call to *exit* (or to *_exit*).

For example, if your program contains the call

```
exit(1)
```

the status is 1.

Passing File Information to Child Processes

The main() Function

If your program uses the exec or spawn functions to create a new process, the new process will normally inherit all of the open file handles created by the original process. Some information, however, about these handles will be lost, including the access mode used to open the file. For example, if your program opens a file for read-only access in binary mode, and then spawns a child process, the child process might corrupt the file by writing to it, or by reading from it in text mode.

To allow child processes to inherit such information about open files, you must link your program with the object file FILEINFO.OBJ.

For example:

```
BCC32 TEST.C \BC5\LIB\FILEINFO.OBJ
```

The file information is passed in the environment variable _C_FILE_INFO. This variable contains encoded binary information. Your program should not attempt to read or modify its value. The child program must have been built with the C++ run-time library to inherit this information correctly.

Other programs can ignore _C_FILE_INFO, and will not inherit file information.

Multithread Programs

See also

32-bit programs can create more than one thread of execution. If your program creates multiple threads, and these threads also use the C++ run-time library, you must use the CW32MT.LIB or CW32MTI library instead.

The multithread libraries provide the following functions which you use to create threads:

_beginthread

_beginthreadNT

The multithread libraries also provide

_endthread a function that terminates threads

_threadid a global variable that contains the current identification number of the thread also known as the *thread ID*).

The header file stddef.h contains the declaration of *_threadid*.

When you compile or link a program that uses multiple threads, you must use the **-tWM** compiler switch. For example:

```
BCC32 -tWM THREAD.C
```

Note: Take special care when using the signal function in a multithread program. The SIGINT, SIGTERM, and SIGBREAK signals can be used only by the main thread (thread one) in a non-Win32 application. When one of these signals occurs, the currently executing thread is suspended, and control transfers to the signal handler (if any) set up by thread one. Other signals can be handled by any thread.

A signal handler should not use C++ run-time library functions, because a semaphore deadlock might occur. Instead, the handler should simply set a flag or post a semaphore, and return immediately.

WinMain

See also

Syntax

```
int PASCAL WinMain(HINSTANCE hCurInstance, HINSTANCE hPrevInstance, LPSTR
  lpCmdLine, int nCmdShow)
```

Description

This function is the main entry point for a Windows application. It must be supplied by the user.

Type	Parameter	Description
HINSTANCE	*hCurInstance*	The instance handle of the application. Each instance of an application has a unique instance handle. It is used as an argument to several Windows functions and can be used to distinguish between multiple instances of a given application.
HINSTANCE	*hPrevInstance*	The handle of the previous instance of this application. This value is NULL if this is the first instance.
LPSTR	*lpCmdLine*	A far pointer to a null-terminated command-line. Specify this value when invoking the application from the program manager or from a call to **WinExec**.
int	*nCmdShow*	An integer that specifies the application's window display. Pass this value to **ShowWindow**.

Under Win32, there are two differences in the values passed through these parameters:

- *hPrevInstance* always returns NULL.
- *lpCmdLine* points to a string containing the entire command line, not just the parameters.

Return Value

The return value from *WinMain* is not currently used by Windows. It is useful during debugging because you can display this value upon termination of your program.

Programming for portability

See also

If you are new to programming, or need to know about moving 16-bit applications to Windows NT or Windows 95, this topic is for you. This topic describes a variety of 16-bit and 32-bit programming topics, including

- Resource script files
- Module definition files
- Import libraries
- The Borland heap manager
- 32-bit Windows programming

In addition to compiling source code and linking .OBJ files, a Windows programmer must compile resource script files, and bind resources to an executable. A Windows programmer must also know about dynamic linking, dynamic link libraries (DLLs), and import libraries. Also, if you are using the Borland C++ IDE, it is helpful to know how to use the Borland project manager which uses project files to automate and manage application building. See the discussion of compiling and linking a Windows program for an illustration of the process of building a Windows application.

Note: The intricacies of designing and developing Windows applications go beyond the scope of this document.

Prologs and epilogs

See also

When you compile a module for Windows, the compiler needs to know what kind of prolog and epilog needs to be created for each of a module's functions. IDE settings and command-line compiler options control the creation of the prolog and epilog. The prolog and epilog perform several duties, including ensuring that the correct data segment is active during callback functions, and marking stack frames for the Windows stack-crawling mechanism.

The prolog/epilog code is automatically generated by the compiler, though various compiler options or IDE settings dictate which sets of instructions are contained in the generated code.

See the following topics for further discussion:

The _export keyword

The _import keyword

Prologs, epilogs, and exports: A Summary

Entry/Exit Code Options

Exporting and importing templates

Compiling and linking a Windows program

See also

These are the steps for compiling and linking a Windows program:

1. Source code is compiled or assembled producing .OBJ files.
2. Module definition files (.DEF) tell the linker what kind of executable you want to produce.
3. Resource Workshop (or some other resource editor) creates resources, like icons or bitmaps. A resource file (.RC) is produced. See the Resource Workshop overview.
4. The .RC file is compiled by a resource compiler or Resource Workshop, and a binary .RES file is output.
5. Linking produces an .EXE file with bound resources.

Resource script files

See also

Windows applications typically use *resources*. Resources are icons, menus, dialog boxes, fonts, cursors, bitmaps, or other user-defined resources. Resources are defined in a file called a resource script file, also known as a resource file. These files have the file name extension .RC.

To make use of resources, you must use the Borland Resource Compiler (BRCC32) to compile your .RC file into a binary format. Resource compilation creates a .RES file. TLINK32 then binds the .RES file to the .EXE file output by the linker. This process also marks the .EXE file as a Windows executable.

Note: See the discussion of BRCC32.EXE.

Module definition files

See also

A module definition (.DEF) file provides information to the linker about the contents and system requirements of a Windows application. This information includes heap and stack size, and code and data characteristics. .DEF files also list functions that are to be made available for other modules (export functions), and functions that are needed from other modules (import functions). Because Borland linkers have other ways of finding out the information contained in a module definition file, module definition files are not always required for Borland's linker to create a Windows application.

Here's the module definition file for the WHELLO example:

```
NAME           WHELLO
DESCRIPTION    'C++ Windows Hello World'
EXETYPE        WINDOWS
CODE           PRELOAD MOVEABLE
DATA           PRELOAD MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      5120
```

Let's take this file apart, statement by statement:

NAME specifies a name for a program. If you want to build a DLL instead of a program, you would use the LIBRARY statement. Every module definition file should have either a NAME statement or a LIBRARY statement, but never both. The name specified must be the same name as the executable file. WINDOWAPI identifies this program as a Windows executable.

DESCRIPTION lets you specify a string that describes your application or library.

EXETYPE marks the executable as a Windows executable. This is necessary for all Windows executables.

CODE describes attributes of the executable's code segment. The PRELOAD option instructs the loader to load this portion of the image when the application is loaded into memory. The MOVEABLE option means Windows can move the code around in memory.

DATA defines the default attributes of data segments. The MULTIPLE option ensures that each instance of the application has its own data segment.

HEAPSIZE specifies the size of the application's local heap.

STACKSIZE specifies the size of the application's local stack. You can't use the STACKSIZE statement to create a stack for a DLL.

Two important statements not used in this .DEF file are the EXPORTS and IMPORTS statements.

The EXPORTS statement lists functions in a program or DLL that will be called by other applications or by Windows. These functions are known as export functions, callbacks, or callback functions. Exported functions are identified by the linker and entered into an export table.

To help you avoid the necessity of creating and maintaining long EXPORTS sections in your module definition files, provides the **_ _export** keyword. Functions flagged with **_ _export** will be identified by the linker and entered into the export table for the module. This is why the WHELLO example has no EXPORT statement in its module definition file.

Note: Prior to Borland C++ 5.0, **_ _export** keyword was required to immediately precede the function name. To help port applications that use a different syntax for function modifiers, Borland C++ now provides the __declspec keyword.

The WHELLO application doesn't have an IMPORTS statement either because the only functions it calls from other modules are those from the Windows Application Program Interface (API); those functions are imported via the automatic inclusion of the IMPORT.LIB or IMPORT32.LIB import libraries. When an application needs to call other external functions, these functions must be listed in the IMPORTS statement, or included via an import library.

Import libraries

See also

When you use DLLs, you must give the linker definitions of the functions you want to import from DLLs. This information temporarily satisfies the external references to the functions called by the compiled code, and tells the Windows loader where to find the functions at run time.

There are two ways to tell the linker about import functions:

You can add an IMPORTS section to the module definition file and list every DLL function that the module will use.

You can include an import library for the DLLs when you link the module.

An import library contains import definitions for some or all of the exported functions for one or more DLLs. A utility called IMPLIB creates import libraries for DLLs. IMPLIB creates import libraries directly from DLLs or from a DLL's module definition files, or from a combination of the two.

Import libraries can be substituted for all or part of the IMPORTS section of a module definition file.

The Borland heap manager

See also

Windows supports dynamic memory allocations on two different heaps: the *global heap* and the *local heap*.

The global heap is a pool of memory available to all applications. Although global memory blocks of any size can be allocated, the global heap is intended only for large memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes, and under the Windows standard and 386 enhanced modes, there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

The local heap is a pool of memory available only to your application. It exists in the upper part of an application's data segment. The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for small memory blocks (256 bytes or less). The default size of the local heap is 4K, but you can change this in your applications .DEF file.

Borland C++ includes a *heap manager* which implements the **new**, **delete**, *malloc*, and *free* functions. The heap manager uses the global heap for all allocations. Because the global heap has a system-wide limit of 8192 memory blocks (which certainly is less than what some applications might require), the Borland C++ heap manager includes a *sub-allocator* algorithm to enhance performance and allow a substantially larger number of blocks to be allocated.

This is how the segment sub-allocator works: When allocating a large block, the heap manager simply allocates a global memory block using the Windows *GlobalAlloc* routine. When allocating a small block, the heap manager allocates a larger global memory block and then divides (sub-allocates) that block into smaller blocks as required. Allocations of small blocks reuse all available sub-allocation space before the heap manager allocates a new global memory block, which, in turn, is further sub-allocated.

The *HeapLimit* variable defines the threshold between small and large heap blocks. *HeapLimit* is set at 64K bytes. The *HeapBlock* variable defines the size the heap manager uses when allocating blocks to be assigned to the sub-allocator. *HeapBlock* is set at 4096 bytes.

32-bit Windows programming

See also

The following topics briefly describe the Win32 and Windows programming environment, and explain how to port your code to this environment. This port makes your code compilable to run on both 16- and 32-bit versions of Windows, and compilable for future processors hosting Windows.

Win32

The Win32 API

Borland C++ 32-bit tools support the production of 32-bit .OBJ and .EXE files in the portable executable (PE) file format, which is the executable file format for Win32 and Windows NT programs. Win32 conforming executables will run unchanged on Windows NT.

Note: See the topic on building Win32 executables for a discussion of 32-bit tool names, options, and libraries.

The Win32 API

See also

The Win32 API widens most of the existing 16-bit Windows API to 32 bits and adds new API calls compatible with Windows NT. The Win32s API is a subset of the Win32 API for Windows NT. Those 16-bit API calls that have been converted to and are callable in the 32-bit environment, and those 32-bit API calls implementable in the 16-bit Windows environment make up the Win32 API.

If a Win32 executable calls any of the Win32 API functions not supported under Win32, appropriate error codes are returned at runtime. Writing applications that conform to the Win32 API, and using the porting tips described under Writing portable Windows code means your application will be portable across 16- and 32-bit Windows environments.

For complete descriptions of Win32 API functions, see the Microsoft Windows documentation.

Win32

See also

Win32 is an operating system extension to Windows 3.1 that provides support for developing and running Windows 32-bit executables. Win32 is a set of DLLs that handle mapping 32-bit application program interface (API) calls to their 16-bit counterparts, a virtual device driver (VxD) to handle memory management, and a revised API called the Win32 API. The DLL and VxD are transparent.

To make sure your code will compile and run under Win32 you should

- Make sure your code adheres to the Win32 API.
- Write portable code using types and macros provided in the windows.h, and windowsx.h files.

See the topic on Writing portable Windows code for some help in writing portable Windows code.

Writing portable Windows code

See also

This topic provides information about portability constructs introduced in Windows 3.1 that will assist you in producing portable Windows code. Explanations of several compiler error and warning messages you might likely see when developing portable code is also included.

- STRICT
- Making your code STRICT compliant
- STRICT conversion hints
- STRICT compliant types, constants, helper macros, and handles
- The UINT and WORD types
- The WINAPI and CALLBACK calling conventions

Existing Windows 16-bit code can be ported to Win32 and Windows NT with minimal changes. Most changes revolve around substituting new macros and types for old, and replacing any 16-bit specific API calls with analogous Win32 API calls. Once these changes have been made, your code can compile and run under 16- or 32-bit Windows.

A compile-time environment variable, STRICT, has been provided to assist you in making your code portable.

STRICT

See also

Windows 3.1 introduced support in windows.h for defining STRICT. Defining STRICT enables strict compiler error checking. For example, if STRICT is not defined, passing an *HWND* to a function that requires an *HDC* will not cause a compiler warning. Define STRICT, and you will get a compiler error.

Using STRICT enables

- Strict handle type checking
- Correct and consistent parameter and return value type declarations
- Fully prototyped type definitions for callback function types (window, dialog, and hook procedures)
- ANSI compliant declaration of COMM, DCB, and COMSTAT structures

STRICT is Windows 3.0 backward compatible. It can be used with the 3.1 WINDOWS.H for creating applications that will run under Windows 3.0.

Defining STRICT will assist you in locating and correcting type incompatibilities that arise when migrating your code to 32 bits, and will aid portability between 16- and 32-bit Windows.

New types, constants, and macros have been provided so you can change your source code to be STRICT compliant. The Table of STRICT compliant types provides a list of the types, macros and handle types that you can use to make your application STRICT compliant.

STRICT compliant types constants, helper macros, and handles

See also

Types and constants	Description
CALLBACK	Use instead of FAR PASCAL in your callback routines (for example, window and dialog procedures).
LPARAM	Declares all 32-bit polymorphic parameters.
LPCSTR	Same as LPSTR, except that is used for read-only string pointers.
LRESULT	Declares all 32-bit polymorphic return values.
UINT	Portable unsigned integer type whose size is determined by the targeted environment. Represents a 16-bit value on Windows 3.1, and a 32 bit value on Win32.
WINAPI	Use instead of FAR PASCAL for API declarations. If you are writing a DLL with exported API entry points, you can use this for the API declarations.
WPARAM	Declares all 16-bit polymorphic parameters.

Macros	Description
FIELDOFFSET*(type, field)*	Calculates the field offsets in a structure. *type* is the structure type, and *field* is the field name.
MAKELP*(sel,off)*	Takes a selector and offset and produces a *FAR VOID**.
MAKELPARAM*(low,high)*	Makes an *LPARAM* out of two 16-bit values.
MAKERESULT*(low,high)*	Makes an *LRESULT* out of two 16-bit values.
OFFSETOF*(lp)*	Extracts the offset of a far pointer and returns a *UINT*.
SELECTOROF*(lp)*	Extracts the selector for a far pointer and returns a *UINT*.

Handles	Description
HACCEL	Accelerator table handle
HDRVR	Driver handle (Windows 3.1 only)
HDWP	*DeferWindowPost()* handle
HFILE	File handle
HGDIOBJ	Generic GDI object handle
HGLOBAL	Global handle
HINSTANCE	Instance handle
HLOCAL	Local handle
HMETAFILE	Metafile handle
HMODULE	Module handle
HRSRC	Resource handle
HTASK	Task handle

Making your code STRICT compliant

See also

This steps will help to make your application STRICT compliant.

1. Decide what code you want to be STRICT compliant. Converting your code to STRICT can be done in stages.
2. Turn on the compiler's highest error/warning level. In the IDE, use the Make|Break Make On options. On the command line, use the **–w** switch to display warnings. You might want to compile at this stage, before taking the next step.
3. `#define STRICT` before including windows.h and compile, or use **–D**STRICT on the command line.

Note: Because of C++ type-safe linking, linking STRICT and non-STRICT modules may cause linker errors in C++ applications.

STRICT conversion hints

See also

This topic describes some common coding practices you should use when converting your code to STRICT compliance.

- Change *HANDLE* to the appropriate specific handle type, for example, *HMODULE*, *HINSTANCE*, and so on.
- Change *WORD* to *UINT* except were you specifically want a 16-bit value on a 32-bit platform.
- Change *WORD* to *WPARAM*.
- Change *LONG* to *LPARAM* or *LRESULT* as appropriate.
- Change *FARPROC* to *WNDPROC*, *DLGPROC*, *HOOKPROC* as appropriate.
- For 16-bit Windows always declare function pointers with the proper function type, rather than *FARPROC*. You'll need to cast function pointers to and from the proper function type when using *MakeProcInstance*, *FreeProcInstance*, and other functions that take or return a *FARPROC*, for example:

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT msg,
                      WPARAM wParam,
                      LPARAM lParam);
                      DLGPROC lpfnDlg;
lpfnDlg=(DLGPROC)MakeProcInstance(DlgProc, hinst);
...
FreeProcInstance((FARPROC)lpfnDlg);
```

- Take special care with *HMODULE*s and *HINSTANCE*s. For the most part, the Kernel module management functions use *HINSTANCE*s, but there are a few APIs that return or accept only *HMODULE*s.
- If you've copied any API function declarations from WINDOWS.H, they may have changed, and your local declaration may be out of date. Remove your local declarations.
- Cast the results of *LocalLock* and *GlobalLock* to the proper kind of data pointer. Parameters to these and other memory management functions should be cast to *LOCALHANDLE* or *GLOBALHANDLE*, as appropriate.
- Cast the result of *GetWindowWord* and *GetWindowLong* and the parameters to *SetWindowWord* and *SetWindowLong*.
- When casting *SendMessage*, *DefWindowProc*, and *SendDlgItemMsg* or any other function that returns an *LRESULT* or *LONG* to a handle of some kind, you must first cast the result to a *UINT*:

```
HBRUSH hbr;
hbr = (HBRUSH)(UINT)SendMessage(hwnd, WM_CTLCOLOR, ..., ...);
```

- The *CreateWindow* and *CreateWindowEx* hmenu parameter is sometimes used to pass an integer control ID. In this case you must cast this to an *HMENU*:

```
HWND hwnd;
int id;
hwnd = CreateWindow("Button", "Ok", BS_PUSHBUTTON,
                    x, y, cx, cy, hwndParent,
                    (HMENU)id,   //Cast required here
                    hinst, NULL);
```

- Polymorphic data types (*WPARAM*, *LPARAM*, *LRESULT*, void *FAR**) should be assigned to variables as soon as possible. You should avoid using them in your own code when the type of the value is known. This will minimize the number of potentially unsafe and non-32-bit-portable casting you will have to do in your code. The macro APIs and message cracker mechanisms provided in windowsx.h will take care of almost all packing and unpacking of these data types in a 32-bit portable way.
- Become familiar with the common compiler warnings and errors that you're likely to encounter as you convert to STRICT.

Some of the most common compiler errors and warnings you might encounter are described under The UINT and WORD types.

See also the description of message crackers.

The UINT and WORD types

See also

The type *UINT* has been created and used extensively in the API to create a data type portable from Windows 3.x. *UINT* is defined as

```
typedef unsigned int UINT;
```

UINT is needed because of the difference in int sizes between 16-bit Windows, and Win32. For 16-bit Windows, int is a 16-bit unsigned integer; for Win32 int is a 32-bit unsigned integer. Use *UINT* to declare integer objects expected to widen from 16 to 32 bits when compiling 32-bit applications.

The type *WORD* is defined as

```
typedef unsigned short WORD;
```

WORD declares a 16-bit value on both 16-bit Windows, and Win32. Use *WORD* to create objects that will remain 16-bits wide across both platforms. Note that because Win32 handles are widened to 32 bits, *WORD* can no longer be used for handles.

The WINAPI and CALLBACK calling conventions

See also

The windows.h macro WINAPI defines the calling convention. WINAPI resolves to the appropriate calling convention for the targeted platform. WINAPI should be used in place of FAR PASCAL.

For example, here is an important change necessary for window procedure definitions. The following is code as it would appear in 16-bit Windows:

```
LONG FAR PASCAL WindowProc(HANDLE hWnd, unsigned message
                           WORD wParam, LONG lParam)
```

Here is the Win32 version:

```
LONG WINAPI WindowProc(HWND hWnd, UINT message
                       UINT wParam, LONG lParam)
```

Using WINAPI allows specifying alternative calling conventions. Currently, Win32 uses _stdcall. The fundamental type unsigned is changed to the more portable *UINT*. *WORD* is also changed to *UINT*, in this case illustrating the expansion of *wParam* to 32 bits. Not making this change to *wParam* will result in application failure during initial window creation.

Use the CALLBACK calling convention in your callback function declarations. This replaces FAR PASCAL.

Extracting message data

See also

In 32-bit Windows code you need to change the way you unpack message data from *lParam* and *wParam*. In Win32 *wParam* grows from 16 to 32 bits in size, while *lParam* remains 32-bits wide. But since *lParam* frequently contains a handle and another value in 16-bit Windows, and a handle grows to 32 bits under Win32, another packing scheme was necessary for *wParam* and *lParam*.

For example, WM_COMMAND is one of the messages affected by the changes to extra parameter size. Under Windows 3.x *wParam* contains a 16-bit identifier, and *lParam* contains both a 16-bit window handle and a 16-bit command.

Under Win32 *lParam* contains the window handle, but nothing else since window handles are now 32 bits. So the 16-bit command is moved from *lParam* to the low-order 16 bits of *wParam* (now 32 bits), with the high order 16 bits of *wParam* containing the identifier. This repacking means changing the way you extract information from these parameters. An easy, portable way of extracting message data is by using message crackers.

Message crackers

See also

Message crackers are a portable way of extracting messages from *wParam* and *lParam*. Depending on your environment (16-bit Windows or Win32) message crackers use an appropriate technique for extracting the message data. Each Windows message has a set of message crackers.

For example, here is the 32-bit version of the WM_COMMAND message crackers:

```
#define GET_WM_COMMAND_ID(wp, lp)               LOWORD(wp)
#define GET_WM_COMMAND_HWND(wp, lp)              (HWND)(lp)
#define GET_WM_COMMAND_CMD(wp, lp)              HIWORD(wp)
#define GET_WM_COMMAND_MPS(id, hwnd, cmd)    \
                          (WPARAM) MAKELONG(id, cmd),
                          (LONG) (hwnd)
```

And here is the 16-bit version of the WM_COMMAND message crackers:

```
#define GET_WM_COMMAND_ID(wp, lp)                (wp)
#define GET_WM_COMMAND_HWND(wp, lp)              (HWND)LOWORD(lp)
#define GET_WM_COMMAND_CMD(wp, lp)               HIWORD(lp)
#define GET_WM_COMMAND_MPS(id, hwnd, cmd)    \
                          (WPARAM)(id), MAKELONG(hwnd, cmd)
```

Using these message-cracker macros will ensure that your message extraction code is portable to either platform.

Porting DOS system calls

See also

Windows 3.0 provided the *DOS3Call* API function for calling DOS file I/O functions. This function, and other INT 21H DOS functions, are replaced in Win32 by named 32-bit calls. See the list of DOS INT 21H calls and their equivalent Win32 API functions.

INT 21 and Win32 equivalent functions

See also

INT 21H function	DOS operation	Win32 API equivalent
0EH	Select disk	*SetCurrentDirectory*
19H	Get current disk	*GetCurrentDirectory*
2AH	Get date	*GetDateAndTime*
2BH	Set date	*SetDateAndTime*
2CH	Get time	*GetDateAndTime*
2DH	Set time	*SetDateAndTime*
36H	Get disk free space	*GetDiskFreeSpace*
39H	Create directory	*CreateDirectory*
3AH	Remove directory	*RemoveDirectory*
3BH	Set current directory	*SetCurrentDirectory*
3CH	Create handle	*CreateFile*
3DH	Open handle	*CreateFile*
3EH	Close handle	*CloseHandle*
3FH	Read handle	*ReadFile*
40H	Write handle	*WriteFile*
41H	Delete file	*DeleteFile*
42H	Move file pointer	*SetFilePointer*
43H	Get file attributes	*GetAttributesFile*
43H	Set file attributes	*SetAttributesFile*
47H	Get current directory	*GetCurrentDirectory*
4EH	Find first file	*FindFirstFile*
4FH	Find next file	*FindNextFile*
56H	Change directory entry	*MoveFile*
57H	Get file date/time	*GetDateAndTimeFile*
57H	Set file date/time	*SetDateAndTimeFile*
59H	Get extended error	*GetLastError*
5AH	Create unique file	*GetTempFileName*
5BH	Create new file	*CreateFile*
5CH	Lock file	*LockFile*
5CH	Unlock file	*UnlockFile*
67H	Set handle count	*SetHandleCount*

Common compiler errors and warnings

See also

This topic describes some of the common compiler errors and warnings you might get when trying to make your application compile cleanly with all messages enabled, and with or with out STRICT defined.

Warning: Call to function *funcname* with no prototype.

This means that a function was used before it was prototyped, or declared. It can also arise when a function that takes no arguments is not prototyped with **void**:

```
void bar(); /* Should be: bar(void) */
void main(void)
{
   bar();
}
```

Warning: Conversion may lose significant digits

This warning results when a value is converted by the compiler, such as from LONG to **int**. You're being warned because you might lose information from this cast. If you're sure there are no information-loss problems, you can suppress this warning with the appropriate explicit cast to the smaller type.

Warning: Function should return a value

This warning means that a function declared to return a value does not return a value. In older, non-ANSI C code, it was common to declare functions that did not return a value with no return type:

```
foo(i)
int i;
{
...
}
```

Functions declared in this manner are treated by the compiler as being declared to return an **int**. If the function does not return anything, it should be declared **void**:

```
void foo(int i)
{
...
}
```

Error: Lvalue required
Error: Type mismatch in parameter

These errors indicate that you are trying to assign or pass a non-pointer type when a pointer type is required. With STRICT defined, all handle types as well as *LRESULT*, *WPARAM*, and *LPARAM* are internally declared as pointer types, so trying to pass an int, *WORD*, or *LONG* as a handle will result in these errors.

These errors should be fixed by properly declaring the non-pointer values you're assigning or passing. In the case of special constants such as (HWND)1 to indicate "insert at bottom" to the window positioning functions, you should use the new macro (such as HWND_BOTTOM). Only in rare cases should you suppress a type mismatch error with a cast. This can often generate incorrect code.

Error: Type mismatch in redeclaration of *paramname*

This error will result if you have inconsistent declarations of a variable, parameter, or function in your source code.

Warning: Conversion may lose significant digits

This warning results when a value is converted by the compiler, such as from LONG to **i**nt. You're being warned because you may lose information from this cast. If you're sure there are no information-loss problems, you can suppress this warning with the appropriate explicit cast to the smaller type.

Warning: Non-portable pointer conversion

This error results when you cast a near pointer or a handle to a 32-bit value such as *LRESULT*, *LPARAM*, *LONG* or *DWORD*. This warning almost always represents a bug, because the high order 16 bits of the value will contain a non-zero value. The compiler first converts the 16-bit near pointer to a 32-bit far pointer by placing the current data segment value in the high 16 bits, then converts this far pointer to the 32-bit value.

To avoid this warning and ensure that a 0 is placed in the high 16 bits, you must first cast the handle to a *UINT*:

```
HWND hwnd;
LRESULT result = (LRESULT)(UINT)hwnd;
```

In cases where you do want the 32-bit value to contain a far pointer, you can **avoid** the warning with an explicit cast to a far pointer:

```
char near* pch;
LPARAM lParam = (LPARAM)(LPSTR)pch;
Error: Size of the type is unknown or zero
```

This error results from trying to change the value of a **void** pointer with + or +=. These typically result from the fact that certain Windows functions that return pointers to arbitrary types (such as *GlobalLock* and *LocalLock*) are defined to return void FAR* rather than LPSTR.

To solve these problems, you should assign the **void*** value to a properly declared variable (with the appropriate cast):

```
BYTE FAR* lpb = (BYTE FAR*)GlobalLock(h);
lpb += sizeof(DWORD);
Error: Not an allowed type
```

This error typically results from trying to dereference a **void** pointer. This usually results from directly using the return value of *GlobalLock* or *LocalLock* as a pointer. To solve this problem, assign the return value to a variable of the appropriate type (with the appropriate cast) before using the pointer:

```
BYTE FAR* lpb = (BYTE FAR*)GlobalLock(h);
```

*lpb = 0;

Warning: Parameter *paramname* is never used

This message can result in callback functions when your code does not use certain parameters. You can either turn off this warning, use `#pragma argsused` to suppress it, or you can omit the name of the parameter in the function definition.

By adhering to the Win32 API, and using STRICT to make code changes you will make your Windows code portable.

Building Win32 executables

See also

You must use the proper tools, switches, libraries, and start-up code to build a Win32 application. The following table lists the compiler (BCC32) and linker (TLINK32) switches, libraries and start-up code commonly needed when linking, and the resulting executable type (.DLL or .EXE).

BCC32 options	TLINK32 option	Libraries	Startup code	Creates this executable type
–tW, –tWE	/Tpe	cw32.lib import32.lib	c0w32.obj	.EXE
–tWD, –tWDE	/Tpd	cw32.lib import32.lib	c0d32.obj	.DLL
–tWC	/Tpe /ap	cw32.lib import32.lib	c0x32.obj	Console .EXE
–tWCD, –tWCDE	/Tpd /ap	cw32.lib import32.lib	c0d32.obj	.DLL

Using Dynamic-Link Libraries

See also

Using DLLs in your applications reduces .EXE file size, conserves system memory, and provides more flexibility in changing, extending, or upgrading your applications. Windows supports both dynamic linking and static linking.

Creating a DLL

You create a DLL in much the same way you create an EXE:

- Source files containing your code are compiled into .OBJ files
- .OBJ files are linked together

The DLL, however, has no *main* function, and is therefore linked differently.

The following topics describe how to write a DLL:

Borland DLLs

DLLs and 16-bit Memory Models

Exporting and Importing Classes

Exporting and Importing Functions

LibMain and DllEntryPoint

WEP (Windows Exit Procedure)

Static Linking

When an application uses a function from a static-link library (for example, the C run-time library), a copy of that function is bound to your application by TLINK at link time. Two applications running simultaneously that use the same function would each have their own copy of that function. It is more efficient, however, if both applications shared a single copy of the function. Dynamic linking provides this capability by resolving your application's references to external functions at run time.

Dynamic linking

When a program uses a function from a DLL, the function code is not linked into the .EXE. Instead, dynamic linking uses a two-step method:

1. At link time, TLINK binds import records (which contain DLL and procedure-location information) to your .EXE. This temporarily satisfies any external references to DLL functions in your code. These import records are supplied by module-definition files or import libraries.
2. At run time, the import-record information is used to locate and bind the DLL functions to your program.

With dynamic linking, your applications are smaller because

- Only one copy of the function code is linked into your application.
- System memory is conserved because DLL code and resources are shared among applications.

DLL

A DLL is an executable library module containing functions or resources for use by applications or other DLLs. A DLL has no *main* function, which is the usual entry point for an application. Instead, a DLLs has multiple entry points, one for each exported function.

When a DLL is loaded by the operating system, the DLL can be shared among multiple applications; one loaded copy of the DLL is all that's necessary.

LibMain and DllEntryPoint

Using Dynamic-link Libraries (DLLs)

Syntax

```
int FAR PASCAL LibMain (HINSTANCE hInstance, WORD wDataSeg, WORD cbHeapSize,
  LPSTR lpCmdLine)
```

Description

You must supply the *LibMain* function for 16-bit programs, or the *DllEntryPoint* (32 bit Windows API) function for 32-bit programs as the main entry point for a DLL.

- For 16-bit programs, Windows calls LibMain once, when the library is first loaded. *LibMain* performs initialization for the DLL.
- For 32-bit programs, Windows calls *DllEntryPoint* each time the DLL is loaded and unloaded (it replaces WEP for 32-bit applications), each time a process attaches to or detaches from the DLL, or each time a thread within the process is created or destroyed.

DLL initialization depends almost entirely on the function of the particular DLL, but might include the following typical tasks:

- Unlocking the data segment with UnlockData, if it has been declared as MOVEABLE.
- Setting up global variables for the DLL, if it uses any.

The initialization code is executed only for the first application using the DLL.

The DLL startup code initializes the local heap automatically; you don't need to include code in *LibMain* to do this.

The following parameters (defined in windows.h) are passed to *LibMain*:

Parameter	Type	Description
HANDLE	hInstance	Instance handle of the DLL
WORD	wDataSeg	Value of the data segment (DS) register
WORD	cbHeapSize	Size of the local heap specified in the module definition file for the DLL.
LPSTR	lpCmdLine	A far pointer to the command line specified when the DLL was loaded. This value is almost always null because DLLs are typically loaded automatically with no parameters. It is possible, however, to supply a command line to a DLL when it is loaded explicitly.

Return Value

On success, *LibMain* returns 1 (successful initialization).

On error, it returns 0 (failure in initialization).

Note: If *LibMain* returns 0, Windows unloads the DLL from memory.

WEP (Windows Exit Procedure)

Using Dynamic-link Libraries (DLLs)

Syntax

```
int FAR PASCAL WEP (int nParameter)
```

where *nParameter* is either

WEP_SYSTEM_EXIT	(indicates that all of Windows is shutting down)
WEP_FREE_DLL	(indicates that only this DLL is being unloaded)

Description

The exit point for a 16-bit DLL is the function WEP (Windows Exit Procedure). This function is not required in a DLL (because the Borland C++ run-time libraries provide a default), but you can supply your own WEP to perform any DLL cleanup before the DLL is unloaded from memory. Windows calls WEP just prior to unloading the DLL.

Under Borland C++, WEP does not need to be exported. Borland C++ defines its own WEP that calls your WEP (if you have defined one), and then performs system cleanup.

Return Value

WEP returns 1 to indicate success. Windows currently does not do anything with this return value.

Exporting and Importing Functions

Using Dynamic-link Libraries (DLLs)

To make your DLL functions accessible to other applications (.EXEs or other DLLs) the function names must be exported. To use exported functions, the function names must be imported.

Exporting Functions

There are two ways to export functions:

- Create a module-definition file with an EXPORTS section listing all functions that will be used by other applications. The IMPDEF tool can help you do this.
- Precede every function name to be exported in the DLL with the keyword _export in the function definition.

A function must be exported from a DLL before it can be imported to another DLL or application.

Importing Functions

If a Windows application module or another DLL uses functions from a DLL, you must tell the linker that you want to import the functions. There are three ways to do this:

- Add an IMPORTS section to the module-definition file and list every DLL function that the module will use.
- Include the import library for the DLLs when you link the module. The IMPLIB tool creates an import library for one or more DLLs.
- Define your function using the _import keyword (32-bit applications only).

DLLs and 16-bit memory models

Using Dynamic-link Libraries (DLLs)

Functions in a DLL are not linked directly into a Windows application. They are called at run time instead. Calls to DLL functions, therefore, will be far calls because the DLL will have a different code segment than the application. The data used by called DLL functions also need to be far.

Suppose you have a Windows application called APP1, a DLL defined by LSOURCE1.C, and a header file for that DLL called lsource1.h. Function f1, which operates on a string, is called by the application.

If you want the function to work correctly regardless of the memory model used to compile the DLL, you need to explicitly make the function and its data far. In the header file lsource1.h, the function prototype would take this form:

```
extern int _export FAR f(char FAR *dstring);
```

In the DLL source LSOURCE1.C, the implementation of the function would take this form:

```
int FAR f1(char far *dstring)
{
.
.
.
}
```

For the application to use the function, the function must be compiled as exportable and then exported. To accomplish this, you can either compile the DLL with all functions exportable (-WD) and list f1 in the EXPORTS section of the module-definition file, or you can flag the function with the _export keyword, as follows:

```
int FAR _export f1(char far *dstring)
{
.
.
.
}
```

If you compile the DLL under the large model (far data, far code), then you don't need to explicitly define the function or its data as far in the DLL. In the header file, the prototype would still take the form shown here because the prototype would need to be correct for a module compiled with a smaller memory model:

```
extern int FAR f1(char FAR *dstring);
```

In the DLL, however, the function could be defined like this:

```
int _export f1(char *dstring)
{

.
.
.
}
```

Remember that before an application can use f1, it has to be imported into the application, either by listing f1 in the IMPORTS section of a module-definition file or by linking with an import library for the DLL.

Exporting and Importing Classes

Using Dynamic-link Libraries (DLLs)

To use classes in a DLL, the class must be exported from the .DLL file and imported by the .EXE file. Conditionalized macro expansion can be used to support both of these circumstances. For example, include something similar to the following code in a header file:

```
#if defined (BUILDING_YOUR_DLL)
   #define _YOURCLASS _export
#elif defined(USING_YOUR_DLL)
   #define _YOURCLASS _import
#else
   #define _YOURCLASS
#endif
```

In your definitions, define your classes like this:

```
class _YOURCLASS class1 {

// ...

};
```

Define BUILD_YOUR_DLL (with the **-D** option, for example) when you are building your DLL. The _YOURCLASS macro will expand to **_import**. Define USE_YOUR_DLL when you are building the .EXE which will use the DLL. The _YOURCLASS macro will expand to **_import**.

See also the discussion on using **_export** with C++ classes.

See also the discussion on exporting and importing templates.

Static Data in 16-bit DLLs

See also Using Dynamic-link Libraries (DLLs)

Through the functions in a DLL, all applications using the DLL have access to the global data in the DLL. In 16-bit DLLs, a particular function will use the same data, regardless of the application that called it (unlike 32-bit DLLs where all data is private to the process). If you want a 16-bit DLL's global data to be protected for use by a single application, you need to write that protection yourself. The DLL itself does not have a mechanism for making global data available to a single application. If you need data to be private for a given caller of a DLL, you need to dynamically allocate the data and manage the access to that data manually. Static data in a 16-bit DLL is global to all callers of a DLL.

Borland DLLs

General forms of compiler and linker command lines that use the DLL versions of the Borland run-time libraries and class libraries are described below.

Here is a 16-bit compile and link using the DLL version of the run-time library:

```
bcc -c -D_RTLDLL -ml source.cpp
tlink -C -Twe c0wl source, , , import crtldll
```

Note that the macro _RTLDLL and the **-ml** switch are use.

Here is the 32-bit version:

```
bcc32 -c -D_RTLDLL source.cpp
tlink32 -Tpe -ap c0x32 source, , , import32 cw32i
```

Here is a 16-bit compile and link using the DLL version of the class library:

```
bcc -c -D_BIDSDLL -ml source.cpp
tlink -C -Twe c0wl source, , , import bidsi crtldll
```

Here is a 32-bit compile and link using the DLL version of the class library:

```
bcc32 -c -D_BIDSDLL source.cpp
tlink32 -Tpe -ap c0x32 source, , , import32 bidsfi cw32i
```

C++ Exception Handling

See also

These topics describe the C++ error-handling mechanism generally referred to as exception handling. The Borland C++ implementation is consistent with the proposed ANSI specification.

- Throwing an Exception
- Handling an Exception
- Exception Specifications
- Constructors and Destructors in Exception Handling
- Unhandled Exceptions
- Setting Exception Handling Options

The C++ language defines a standard for exception handling. The standard insures that the power of object-oriented design is supported throughout your program. An especially strong feature of the standard is the availability of virtual functions and the use of objects to define exceptions. Virtual functions guarantee a minimum of runtime overhead--zero additional program overhead if no exceptions are thrown.

In accordance with the ANSI/ISO working paper specification, Borland C++ supports the termination exception-handling model. When an abnormal situation arises at runtime, the program should terminate. However, throwing an exception allows you to gather information at the throw point that could be useful in diagnosing the causes which led to failure. You can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled, meaning that the cause of failure is generated from within the program. An event such as Control-C (which is generated from outside the program) is not considered to be an exception.

When the program encounters an abnormal situation for which it is not designed, you may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.

The exception-handling mechanism requires the use of three keywords: try, catch, and throw. The *try-block* specified by **try** must be followed immediately by the *handler* specified by **catch**. If an exception is thrown in the *try-block*, program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of program.

Although C++ allows an exception to be of any type, it is useful to make exceptions objects. The exception object is treated exactly the way any object would. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This is information that the program user will want to know when the program encounters some anomaly at runtime.

Throwing an Exception

See also Example C++ Exception Handling

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is tranferred to the handler

If no handler is found, the program will call the *terminate* function. If no exceptions are thrown, the program executes in the normal fashion.

A throw expression is also referred to as a throw-point. You can specify whether an exception may be thrown by using one of the following syntax specifications:

When an exception occurs, the throw expression initializes a temporary object of the type **T** (to match the type of argument *arg*) used in *throw*(**T** *arg*). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

Example 1

```
throw throw_object;
```

This example specifies that throw_object is to be passed to a handler.

Example 2

```
throw;
```

This example simply specifies that the last exception thrown is to be thrown again. An exception must currently exist. Otherwise, terminate is called.

Example 3

```
void my_func1() throw (A, B)
{
    // Body of function.
}
```

This example specifies a list of exceptions that *my_func1* can throw. No other exceptions will propagate out of *my_func1*. If an exception other than *A* or *B* is generated within *my_func1*, it is considered to be an unexpected exception and program control will be transferred to the *unexpected* function.

Example 4

```
void my_func2() throw ()
{
     // Body of this function.
}
```

The final case specifies that *my_func2* will throw no exceptions. If any function in the body of *my_func2* throws an exception, such an exception will not exist beyond the body of *my_func2*.

Handling an Exception

See also Examples C++ Exception Handling

The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the try-block. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call *terminate*.

Exception handlers are evaluated in the order that they are encountered. An exception is caught when its type matches the type in the **catch** statement. Once a type match is made, program control is transferred to the handler. The stack will have been unwound upon entering the handler. The handler specifies what actions should be taken to deal with the program anomaly.

A **goto** statement can be used to transfer program control out of a handler but such a statement can never be used to enter a handler.

After the handler has executed, the program can continue at the point after the last handler for the current try-block. No other handlers are evaluated for the current exception.

Examples

Example 1

Example 2

Example 1

```
try {
   // Include any code that might throw an exception
}
catch (T X) // Provide a handler for each exception that might be thrown
  above
{
  // Take some actions
}
```

This example is specifically defined to handle an object of type **T**. If the argument is **T, T&, const T**, or **const T&**, the handler will accept an object of type **X** if any of the following are true:

- **T** and **X** are of the same type
- **T** is an accessible base class for **X** in the throw expression
- **T** is a pointer type and **X** is a pointer type that can be converted to **T** by a standard pointer conversion in the throw expression

Example 2

```
try {
   // Include any code that might throw an exception
}
catch ( ... )
{
  // Take some actions
}
```

The statement **catch** (...) will handle any exception, regardless of type. This statement, if used, must be the last handler for its try-block.

Exception Specifications

See also Examples C++ Exception Handling

The C++ language makes it possible for you to specify any exceptions that a function can throw. This *exception specification* can be used as a suffix to the function declaration. The syntax for exception specification is as follows:

```
exception-specification:
   throw (type-id-listopt)
   type-id-list:
   type-id
   type-id-list, type-id
```

The function suffix is not considered to be part of the function's type. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return and argument types. Therefore, the following is legal:

```
void f2(void) throw();       // Should not throw exceptions
void f3(void) throw (BETA); // Should only throw BETA objects
void (* fptr)();             // Pointer to a function returning void
fptr = f2;
fptr = f3;
```

Extreme care should be taken when overriding virtual functions. Again, because the exception specification is not considered part of the function type, it is possible to violate the program design.

If an exception is thrown which is not listed in the exception specification, the unexpected function will be called.

Examples

Example 1

Example 2

Example 3

Example 1

In the following example, the derived class *BETA::vfunc* is defined so that it should not throw any exceptions--a departure from the original function declaration.

```
class ALPHA {
public:
   struct ALPHA_ERR {};
   virtual void vfunc(void) throw (ALPHA_ERR) {}; // Exception specification
};

class BETA : public ALPHA {
   void vfunc(void) throw() {};  // Exception specification is changed
};
```

The following are examples of functions with exception specifications.

```
void f1();                // The function can throw any exception

void f2() throw();        // Should not throw any exceptions

void f3() throw( A, B* ); // Can throw exceptions publicly derived from A,
                          // or a pointer to publicly derived B
```

The definition and all declarations of such a function must have an exception specification containing the same set of type-id's. If a function throws an exception not listed in its specification, the program will call *unexpected*. This is a runtime issue--it will not be flagged at compile time. Therefore, care must be taken to handle any exceptions which can be thrown by elements called within a function.

Example 2

```
// HOW TO MAKE EXCEPTION-SPECIFICATIONS AND HANDLE ALL EXCEPTIONS
#include <iostream.h>

// EXCEPTION DECLARATIONS
class Alpha {
   // Include something that shows why you chose to throw this exception.
};
Alpha alpha_inst;

class Beta {
   // Include something that shows why you chose to throw this exception.
};
Beta beta_inst;

// THROW ONLY Alpha OR Beta TYPE OBJECTS
void f3(char c) throw (Alpha, Beta) {
   cout << "f3() was called" << endl;
   if (c == 'a')
      throw( alpha_inst );
   if (c == 'b')
      throw( beta_inst );
   else ; // DO NOTHING WITH OTHER CHARACTERS
   }

// SHOULD NOT THROW EXCEPTIONS
void f2(char ch) throw() {
   try {                        // WRAP ALL CODE IN A TRY-BLOCK
      cout << "f2() was called" << endl;
      f3(ch);
      }
   // HERE ARE HANDLERS FOR THE EXCEPTIONS WE KNOW COULD BE THROWN
   catch (Alpha& alpha_inst) { cout << "Caught Alpha exception.";}
   catch (Beta& beta_inst) { cout << "Caught Beta exception.";}

   // IF THE CODE IS MODIFIED LATER SO THAT SOME
   // OTHER EXCEPTION IS THROWN, IT IS HANDLED HERE
   // AND WE AVOID VIOLATING THE f2() THROW SPECIFICATION
   catch ( ... ) {
      // BUT, WE POST OURSELVES A WARNING MESSAGE.
      cout << "Warning: f2() has elements with exceptions!" << endl;
      }
   }

int main(void) {
   char trigger;

   try {
      cout << "Input a character:";
      cin >> trigger;
      f2(trigger);
      cout << "\nSuccess.";
      return 0;  // WE GET HERE ONLY IF EVERYTHING EXECUTES WELL.
      }
   catch ( ... ) {
      cout << "Need more handlers!";
```

```
    return 1;
    }
}
```

Sample output when ‘a’ is the input:

Input a character: a

f2() was called

f3() was called

Caught Alpha exception.

Success.

Example 3

The following examples illustrate the sequence of events which can occur when *unexpected* is called.

Program behavior when a function is registered with set_unexpected():

```
unexpected()  // CALLED AUTOMATICALLY
   |
   |
   |                 // DEFINE YOUR UNEXPECTED HANDLER
   |                unexpected_function my_unexpected( void )
   |                {
   |                   // DEFINE ACTIONS TO TAKE
   |                   // POSSIBLY MAKE ADJUSTMENTS
   |                }
   |
   |                // REGISTER YOUR HANDLER
   |                set_unexpected( my_unexpected );
   |
my_unexpected();
```

Program behavior when no function is registered with set_unexpected() but there is a function registered with set_terminate():

```
unexpected()  // CALLED AUTOMATICALLY
   |
   |
   |           // DEFINE YOUR TERMINATION SCHEME
   |           terminate_function my_terminate( void )
   |              {
   |              // TAKE ACTIONS BEFORE TERMINATING
   |              // SHOULD NOT THROW EXCEPTIONS
   |              exit(1); // MUST END SOMEHOW.
   |              }
   |
   |           // REGISTER YOUR TERMINATION FUNCTION
   |           set_terminate( my_terminate )
   |
terminate()
   |
my_terminate()
// PROGRAM ENDS.
```

Constructors and Destructors in Exception Handling

See also C++ Exception Handling

When an exception is thrown, the copy constructor is called for the exception. The copy constructor is used to initialize a temporary object at the throw point. Other copies may be generated by the program.

When program flow is interrupted by an exception, destructors are called for all automatic objects which were constructed since the beginning of the the try-block was entered. If the exception was thrown during construction of some object, destructors will be called only for those objects which were fully constructed. For example, if an array of objects was under contruction when an exception was thrown, destructors will be called only for the array elements which were already fully constructed.

The effect of calling destructors for automatic objects is referred to as stack unwinding. Stack unwinding always occurs. Destructors are called by default but the default can be switched off.

Unhandled Exceptions

See also C++ Exception Handling

If an exception is thrown and no handler is found it, the program will call the *terminate* function. This example illustrates the series of events that can occur when the program encounters an exception for which no handler can be found.

```
terminate();
   .
   .
   .
abort();  // PROGRAM ENDS.
```

Setting Exception Handling Options

See also C++ Exception Handling

IDE Setting	Command-Line Switch
Enable exception handling	-x
Enable destructor cleanup	-xd
Enable throwing exceptions from a DLL	-xds
Enable exception location information	-xp

C-Based Structured Exceptions

See also

Borland C++ provides support for program development that makes use of structured exceptions. You can compile and link a C source file that contains an implementation of structured exceptions. In a C program, the ANSI-compatible keywords used to implement structured exceptions are __*except*, __finally, and __try.

Note: The __*finally* and __*try* keywords can appear only in C programs.

try-except Exception-Handling Syntax

For try-except exception-handling implementations the syntax is as follows:

```
try-block:
  __try compound-statement (in a C module)
  try compound-statement (in a C++ module)
handler:
  __except (expression) compound-statement
```

try-finally Termination Syntax

For try-finally termination implementations the syntax is as follows:

```
try-block:
  __try compound-statement
termination:
  __finally compound-statement
```

See your Win32 documentation for additional details on the implementation of structured exceptions for 16- and 32-bit platforms.

Using C-Based Exceptions in C++ Programs

See also Example

Borland C++ allows substantial interaction between C and C++ error handling mechanisms. The following interactions are supported:

- C structured exceptions can be used in C++ programs.
- C++ exceptions cannot be used in a C program because C++ exceptions require that their handler be specified by the **catch** keyword and **catch** is not allowed in a C program.
- An exception generated by a call to the *RaiseException* function is handled by a **try**/**_ _except** or **_ _try**/**_ _except** block. All handlers of **try**/**catch** blocks are ignored when *RaiseException* is called.
- The following C exception helper functions can be used in a C and C++ programs:
  - *GetExceptionCode*
  - *GetExceptionInformation*
  - *SetUnhandledExceptionFilter*
  - *UnhandledExceptionFilter*

Borland C++ does not enforce the use of *UnhandledExceptionFilter* function only in the except filter of **_ _try**/**_ _except** or **try**/**_ _except** blocks. However, program behavior is undefined when this function is called outside of the **_ _try**/**_except** or **try**/**_ _except** block.

The full functionality of an **_ _except** block is allowed in C++. If an exception is generated in a C module, it is possible to provide a handler-block in a separate calling C++ module. If no handler is found in the calling module, the default action is to terminate the program.

If a handler can be found for the generated structured exception, the following actions can be taken:

- execute the actions specified by the handler
- ignore the generated exception and resume program execution
- continue the search for some other handler (regenerate the exception)

These actions are consistent with the design of structured exceptions.

The **_ _try**/**_ _finally** ensures that the code in the **_ _finally** block is executed no matter how the flow within the **_ _try** exits. The **_ _finally** keyword is not allowed in a C++ program and the **_ _try**/**_ _finally** block is not supported in a C++ program.

Even though the **_ _try**/**_ _finally** block is not supported in a C++ program, a C-based exception generated by the operating system or the program can still result in proper stack unwinding by using local objects within destructors. Any module compiled with the **-xd** compiler option will have destructors invoked for all objects with **auto** storage. Stack unwinding occurs from the point where the exception is thrown to the point where the exception is caught.

C-Based Exceptions in C++ Programs Example

```
/* In PROG.C */
void func(void) {

   .
   .
   .
   /* generate an exception */
   RaiseException( /* specifiy your arguments */ );

   .
   .
   .
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
   try
   {            // test for C++ exceptions
      try
      {         // test for structured exceptions
         func();
      }
      __except( /* filter-expression */ )
      {
      cout << "A structured exception was generated.";

   .
   .
   .
      /* specify actions to take for this structured exception */
      return -1;
      }
      return 0;
   }
   catch ( ... )
   {
   // handler for any C++ exception
   cout << "A C++ exception was thrown.";
   return 1;
   }
}
```

International API (16-bit)

See Also

Borland C++ provides support for international program development. For 16-bit applications, support is provided only for the Great Britain and United States English, French, and German locales.

Note: For 32-bit applications, full support is provided for multibyte and Unicode character processing as well as locale support via code page selection. See International API Routines. for a description of the 32-bit implementation.

To illustrate the effects of selecting different locales, Borland C++ provides a sample ObjectWindows application named INTLDEMO.EXE.

The LOCALE.BLL file is installed in BC5\BIN directory.

Support for these locales is contained in the LOCALE.BLL library. By default, the **C** locale is in effect. A call to setlocale will

- dynamically link the LOCALE.BLL library to your program.
- enable a locale-specific module.
- specify which character set (also called *code page* or *code set*) to use with the locale

You can query the locale settings by using localeconv and *setlocale* functions.

If the call to *setlocale* can be resolved, several character-handling functions change their behavior.

Note: Because the 16-bit version of the Borland C++ international API supports only 8-bit characters (thereby enabling recognition of as many as 256 characters), only single-byte character-handling functions are affected. For a list of affected functions, see International API Routines.

In your code, you must #define _ _USELOCALES_ _ to have the locale-sensitive functions available. Otherwise, only the **C** type locale macros will be used.

See Also

International API Routines.InternationalAPIRoutines16bit

International API Routines.InternationalAPIRoutines

International API Sample Program

International API Sample Program (ANSI character set)

See Also

To illustrate the effects of selecting different locales, Borland C++ provides a sample ObjectWindows application named INTLDEMO.EXE

INTLDEMO is located in BC5\EXAMPLES\OWLAPPS\INTLDEMO which includes all source code and a project file.

Note: You must install ObjectWindows before you can build INTLDEMO.EXE.

It demonstrates how the setlocale function can produce an internationally aware Windows application.

The program lets you choose to display the user interface in English, French, or German.

ANSI Character Set

INTLDEMO displays the Windows ANSI character set (also referred to as the WIN 1252 character set) in a 16-by-16 character grid.

Characters are highlighted according to the selections under the Locale and Classification menus. When you run INTLDEMO, the screen shows the default **C** locale, and the default classification is isalpha. The highlighted characters are A to Z and a to z. If you select another locale such as French, the accented versions of the characters are also highlighted. Various combinations of locale and classification can be illustrated by selecting the appropriate menu items.

File Menu

Choose File|List to open a dialog box that demonstrates the effects of the locale on

- collation sequences
- date and time functions

The files in the current directory are sorted according to the current locale. File dates and times display according to the conventions in the language of the selected locale.

File names can be switched between upper and lower case to demonstrate the effects of the toupper and tolower functions in the current locale. The dialog also illustrates the effects of the BWCCIntlInit function on the OK, Cancel, and Help buttons of the dialog.

You can select any file in the directory to view and sort its contents according to the locale collation sequence currently selected.

Conventions Menu

Choose Conventions|Show to display the results of calling the localeconv function for the current locale. For example, if you select the French locale, the international currency symbol becomes FRF and the currency symbol becomes F.

Note: If you keep the Conversions window open as you select a new language or locale, its values are updated immediately.

Language Menu

Use the Language menu to change the language of the user interface. This feature uses ObjectWindows to associate windows interface elements to a module, in this case a .DLL that contains the resources in a particular language.

When you change the language, the program loads a new language DLL and then reloads the interface elements from the new DLL.

Note: Date and times in the File List dialog are not affected by the change in language, but by the choice of locale.

Classification Menu

Use the Classification menu to see a list of the locale-sensitive *is....()* functions. Selecting one of these items will highlight characters in the main window that return true for the selected function in the current locale.

See Also

Classification Routines

International API Routines.

The Inline Assembler (BASM)

With the Borland C++ inline assembler, you can write 8086/8087 and 80286/80287 assembler code directly inside your C and C++ programs.

Index to Inline Assembler Help

asm
assembler directive
expression classes
expression operators
expression symbols
expression types
expressions
instruction opcodes
labels
numeric constants
operands
predefined type symbols
prefix opcodes
register symbols
relocatable expressions
reserved words
string constants

Using the Inline Assembler

You access the inline assembler through assembler statements (with the asm directive).

Expressions

The Borland C++ inline assembler operands are expressions. The basic elements of an expression are constants, registers, symbols, and operators.

The inline assembler divides expressions into three classes:

registers
memory references
immediate values

Symbols

The inline assembler provides access to almost all C++ symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions.

In addition to any currently declared C++ types, the inline assembler provides several predefined type symbols.

Constants

The Borland C++ inline assembler supports two types of constants:

numeric constants
string constants

Opcodes, Operators, and Directives

The Borland C++ inline assembler supports:

- All 8086/8087 and 80286/80287 instructions
- Opcodes

- Most Turbo Assembler expression operators
- Turbo Assembler's define byte, define word, and define double word directives (DB, DW, and DD)

The inline assembler also implements a large subset of the syntax supported by Turbo Assembler and Microsoft's Macro Assembler.

Note: If you plan to do a lot of assembly language programming, you can use TASM (the stand-alone assembler) to create entire modules coded in assembly language. These .OBJ file modules can then be linked into your C++ applications. TASM, sold separately from Borland C++, supports all 80*x*86 processors and contains complete documentation on the assembler and assembly language.

Inline Assembler Operands

Inline assembler operands are expressions made up of a combination of constants, registers, symbols, and operators.

Although inline assembler expressions are built using the same basic principles as C++ expressions, there are some important differences.

The inline assembler:

- Recognizes its own set of reserved words.
- Evaluates all expressions as 32-bit integer values.
- Interprets variable references as the address of the variable. (C++ interprets them as the variable's contents.)

Also, all inline assembler expressions must resolve to a constant value.

Inline Assembler Expressions

The Borland C++ inline assembler operands are expressions.

Inline assembler expressions are built from expression elements (constants, registers, and symbols) and operators. Each expression has an associated expression class and expression type.

Evaluation

The inline assembler evaluates all expressions as 32-bit integer values. It does not support floating-point and string values, except string constants.

Note: The most important difference between C++ expressions and inline assembler expressions is that all inline assembler expressions must resolve to a constant value (a value that can be computed at compile time).

Inline Assembler Expression Classes

The inline assembler divides expressions into three classes:

registers
memory references
immediate values

Inline Assembler Expression Operators

This table lists the inline assembler's expression operators in decreasing order of precedence. The operators within each category have equal precedence.

Category	Operator	What it is (or does)
Highest	(...)	Subexpression
	[...]	Memory reference
	.	Structure member selector
Unary	HIGH	Returns high-order 8 bits
	LOW	Returns low-order 8 bits
	+	Unary plus
	-	Unary minus
	:	Segment override
	OFFSET	Returns offset part
	SEG	Returns segment part
	TYPE	Returns type (byte size)
	PTR	Typecast
	*	Multiplication
	/	Integer division
	MOD	Integer modulus (remainder)
	SHL	Logical shift left
	SHR	Logical shift right
Additive	+	Binary addition
	-	Binary subtraction
Bitwise	NOT	Bitwise negation
	AND	Bitwise AND
	OR	Bitwise OR
	XOR	Bitwise Exclusive OR

Inline Assembler Expression Types

Every inline assembler expression has an associated type.

This type is a size, because the inline assembler regards the type of an expression as the size of its memory location.

The inline assembler performs type checking whenever possible; an error results if the type check fails.

You can use a typecast to change the type of a memory reference. For example, all of these refer to the first (least significant) byte of the OutBufPtr variable:

```
asm {
  MOV     DL,BYTE PTR OutBufPtr
  MOV     DL,Byte(OutBufPtr)
  MOV     DL,OutBufPtr.Byte
}
```

In some cases, a memory reference is untyped (it has no associated type); for example, an immediate value enclosed in square brackets:

```
asm {
  MOV     AL,[100H]
  MOV     BX,[100H]
}
```

The inline assembler permits both of these instructions because

- the expression `[100H]` has no associated type (it just means "the contents of address 100H in the data segment") and
- the type can be determined from the first operand (byte for AL, word for BX).

In cases where the type can't be determined from another operand, the inline assembler requires an explicit typecast, like this:

```
asm {
  INC     BYTE PTR [100H]
  IMUL    WORD PTR [100H]
}
```

Inline Assembler Predefined Type Symbols

In addition to any currently declared C++ types, the inline assembler provides these predefined type symbols.

Symbol	Type
BYTE	1
WORD	2
DWORD	4
NEAR	0FFFEH
FAR	0FFFFH

Inline Assembler NEAR and FAR Pseudo-Types

The NEAR and FAR pseudo-types are used by procedure and function symbols to indicate their call model.

You can use NEAR and FAR in typecasts just like other symbols. For example, if FarFunction is a FAR function:

```
void far farFunction (void);
```

and you're writing inline assembler code in the same module as FarFunction, you can use the more efficient NEAR call instruction to call FarFunction:

```
asm {
  PUSH    CS
  CALL    NEAR PTR FarFunction
}
```

Inline Assembler Register Expressions

An inline assembler expression that consists solely of a register name is a register expression.

Examples of register expressions are AX, CL, DI, and ES.

Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Inline Assembler Memory Reference Expressions

Inline assembler expressions that denote memory locations are memory references.
C++'s labels, variables, typed constants, procedures, and functions belong to this category.
Memory references are further classified as either relocatable or absolute expressions.

Inline Assembler Immediate Value Expressions

Inline assembler expressions that aren't registers and aren't associated with memory locations are immediate values.

This group includes C++'s untyped constants.

Immediate values are further classified as either relocatable or absolute expressions.

Inline Assembler Relocatable vs. Absolute Expressions

Typically, an expression that refers to a label, variable, procedure, or function is relocatable, and an expression that operates solely on constants is absolute.

- A relocatable expression denotes a value that requires relocation at link time.
- An absolute expression denotes a value that requires no such relocation.

(Relocation is the process by which the linker assigns absolute addresses to symbols.)

At compile time, the compiler does not know the final address of a label, variable, procedure, or function.

The final address does not become known until link time, when the linker assigns a specific absolute address to the symbol.

The inline assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Inline Assembler Register Symbols

In the Borland C++ inline assembler, the following reserved symbols denote CPU registers:

Symbols	Registers
AX BX CX DX	16-bit general purpose
AL BL CL DL	8-bit low registers
AH BH CH DH	8-bit high registers
SP BP SI DI	16-bit pointer or index
CS DS SS ES	16-bit segment registers
ST	8087 register stack

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands.

Register Indexing

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing.

These are valid index register combinations:

```
[BP]
[BP+DI]
[BP+SI]
[BX]
[BX+DI]
[BX+SI]
[DI]
[SI]
```

Segment Overriding

The segment registers (ES, CS, SS, and DS) can be used in conjunction with the colon (:) segment override operator to indicate a different segment than the one the processor selects by default.

For example:

```
asm mov AX, 0xb000
asm mov ES, AX
asm mov SI, WORDPTR es:[0]
```

Inline Assembler Expression Symbols

The inline assembler provides access to almost all C++ symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions.

The inline assembler also provides several predefined type symbols.

Symbol	Value	Class	Type
Label	Address of label	Memory	Short
Constant	Value of constant	Immediate	0
Type	0	Memory	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory	Size of type
Function	Address of function	Memory	Near or Far

Symbols that can NOT be used

The following symbols can NOT be used in inline assembler expressions:

- String, floating-point, and set constants
- Functions declared with the inline modifier
- Labels that aren't declared in the current block

Local Variables

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to SS:BP.

The value of a local variable symbol is its signed offset from SS:BP.

The inline assembler automatically adds [BP] in references to local variables.

Scope

A scope is provided by type, field, and variable symbols of a structure or object type.

Operator

Some symbols, such as structure types and variables, have a scope that can be accessed using the structure member selector (.) operator.

Type Identifier

You can use a type identifier to construct variables "on the fly".

ST(x) Register Symbol

The symbol ST denotes the topmost register on the 8087 floating-point register stack.

Each of the eight floating-point registers can be referred to using `ST(x)`, where X is a constant between 0 and 7, indicating the distance from the top of the register stack.

Inline Assembler String Constants

In inline assembler statements, string constants must be enclosed in single or double quotes.

Two consecutive quotes of the same type as the enclosing quotes count as only one character.

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string.

When not in a DB directive, a string constant can be no longer than four characters, and denotes a numeric value which can participate in an expression.

If the string is shorter than four characters, the leftmost (first) character(s) are assumed to be 0 (zero).

Here are some examples of string constants and their corresponding numeric values:

String Constant	Numeric Value
`'a'`	00000061H
'ba'	00006261H
`'cba'`	00636261H
`'dcba'`	64636261H
`'a '`	00006120H
`'  a'`	20202061H
`'a'*2`	000000E2H
`'a'-'A'`	00000020H
`NOT 'a'`	FFFFFF9EH

Inline Assembler Numeric Constants

Inline assembler numeric constants must be integers between -2,147,483,648 and 4,294,967,295, and they must start with one of the digits 0 through 9 or a $ character.

By default, numeric constants use decimal (base 10) notation, but the inline assembler supports binary (base 2), octal (base 8), and hexadecimal (base 16) notations as well.

To select this notation	**Write a...**
Binary	letter `B` after the number
Octal	letter `O` after the number
Hexadecimal	letter `H` after the number, or a `$` before the number

C++ expressions allow only decimal notation and hexadecimal notation (using a $ prefix); they don't support the B, O, and H suffixes.

When you write a hexadecimal constant using the H suffix, and the first significant digit is one of the hexadecimal digits A through F, an extra zero (0) in front of the number is required.

Examples:

```
0BAD4H Hexadecimal constant
$BAD4  Hexadecimal constant
BAD4H  Identifier (it starts with a letter B, not a digit)
```

Inline Assembler Reserved Words

Within operands, the following reserved words have a predefined meaning to the inline assembler:

Reserved Words Table

AH
AL
AND
AX
BH
BL
BP
BX
BYTE
CH
CL
CS
CX
DH
DI
DL
DS
DWORD
DX
ES
FAR
HIGH
LOW
MOD
NEAR
NOT
OFFSET
OR
PTR
SEG
SHL
SHR
SI
SP
SS
ST
TYPE
WORD
XOR

These reserved words always take precedence over user-defined identifiers.

Inline Assembler Prefix Opcodes

The inline assembler supports the following prefix opcodes:

Opcode	What It Means
LOCK	Bus lock
REP	Repeat string operation
REPE	Repeat string operation while equal
REPZ	Repeat string operation while 0
REPNE	Repeat string operation while not equal
REPNZ	Repeat string operation while not 0

An assembler instruction can be prefixed by zero, one, two, or three of these opcodes. Any more than three prefix opcodes won't make sense.

If you specify a prefix opcode without an instruction opcodes in the same statement, the prefix opcode affects the instruction opcode in the next assembler statement.

Because some 80x86 processors do not handle all combinations correctly, ordering in multiple prefix opcodes is important.

Inline Assembler Instruction Opcodes

The inline assembler supports all 8086/8087 and 80286/80287 instruction opcodes.

For complete descriptions of the instruction opcodes, refer to your 80x86 and 80x87 manuals.

See Also

inline assembler RET instructions

automatic jump sizing

Inline Assembler RET Instructions

Depending on the call model of the current procedure or function, RET generates either a near return or a far return machine-code instruction.

Inline Assembler Jump Optimization

The inline assembler optimizes jump instructions by automatically selecting the shortest, most efficient form of a jump instruction.

When the target is a label (not a procedure or function), this automatic jump sizing applies to JMP and to all conditional jump instructions.

Opcode	Distance to Target Label	Inline Assembler Generates
JMP	Within -128 to 127 bytes	Short jump
	NOT within -127 to 128 bytes	Near jump
Conditional	Within -128 to 127 bytes	Short jump
Jumps	NOT within -127 to 128 bytes	Short jump

Jumps to the entry points of procedures and functions are always either near or far, but never short.

Conditional jumps to procedures and functions are not allowed.

Inline Assembler DB, DW, and DD Directives

The inline assembler supports three assembler directives: DB (define byte), DW (define word), and DD (define double word).

Dir	Operand Type	Value Range	Inline Assembler Generates
DB	Constant expression	-128 to 255	1 byte
	Character string	Any length	Sequence of bytes corresponding to ASCII code of each character
DW	Constant expression	-32,768 to 65,535	1 word
	Address expression		Near pointer (offset word)
DD	Constant expression	-2,147,483,648 to 4,294,967,295	1 double word
	Address expression		Far pointer (offset word followed by segment word)

The data generated by the DB, DW, and DD directives is always stored in the code segment.

To generate uninitialized or initialized data in the data segment, use normal C++ declarations.

Here are some examples of DB, DW, and DD directives:

Dir	Operand	Result
DB	0FFH	One byte
DB	0,99	Two bytes
DB	'Hello...', 0DH,0AH	String + CR/LF
DB	12,"Borland C++"	C++-style string
DW	0FFFFH	One word
DW	0,9999	Two words
DW	'A'	Same as DB 'A',0
DW	'BA'	Same as DB 'A','B'
DW	MyVar	Offset of MyVar
DW	MyProc	Offset of MyProc
DD	0FFFFFFFFH	One double-word
DD	0,999999999	Two double-words
DD	'A'	Same as DB 'A',0,0,0
DD	'DCBA'	Same as DB 'A','B','C','D'
DD	MyVar	Pointer to MyVar
DD	MyProc	Pointer to MyProc

The only kind of symbol that can be defined in an inline assembler statement is a label. All variables must be declared using C or C++ syntax.

Inline Assembler Statement

The syntax of an assembler statement is

```
opcode operand <operand operand   >; or Newline
```

where

- "opcode" is an assembler instruction opcode
- "operand" is an assembler expression
- ";or Newline" is a semicolon or a new line, either of which signals the end of the asm statement.

Comments are allowed between assembler statements, but not within them.

Inline Assembler Labels

Labels consist of a label identifier and a colon before a statement.

Inline Assembler Subexpression (...)

The expression within the parentheses is a subexpression.

```
(Expression)
```

Subexpressions are evaluated completely before they are treated as a single expression element.

Another expression can precede the expression within the parentheses. In this case, the result becomes the sum of the values of the two expressions, with the type of the first expression.

Inline Assembler Memory Reference Operator [...]

The expression within brackets refers to a memory location.

```
[Expression]
```

This memory reference expression is evaluated completely prior to being treated as a single expression element.

To indicate CPU register indexing, you can use the plus (+) operator to combine the memory reference expression with the BX, BP, SI, or DI registers.

Another expression can precede the memory reference expression. In this case, the result becomes the sum of the values of the two expressions, with the type of the first expression.

Result

Always a memory reference.

Inline Assembler Structure Member Operator (xxx . yyy)

The second expression is a member of the structure identified by the first expression.

```
Expression1.Expression2
```

Result
Sum of the two expressions.

Result Type
Type of the second expression.

Symbols belonging to the scope identified by the first expression can be accessed in the second expression.

Inline Assembler HIGH Operator

HIGH returns the high-order 8 bits of the word-sized expression following the operator.

```
HIGH Expression
```

The expression must be an absolute immediate value.

Inline Assembler LOW Operator

LOW returns the low-order 8 bits of the word-sized expression following the operator.

```
LOW Expression
```

The expression must be an absolute immediate value.

Inline Assembler Unary Plus (+...)

Unary plus returns the expression following the plus with no changes.

```
+Expression
```

The expression must be an absolute immediate value.

Inline Assembler Unary Minus (-...)

Unary minus returns the negated value of the expression following the minus.

```
-Expression
```

The expression must be an absolute immediate value.

Inline Assembler Segment Override Operator (: ...)

This operator instructs the assembler that the expression after the colon belongs to the segment given by the segment register name before the colon (CS, DS, SS, or ES).

```
XX:Expression
```

where XX = CS, DS, SS, or ES.

Result

A memory reference with the value of the expression after the colon.

When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction.

This ensures that the indicated segment is selected.

Inline Assembler OFFSET Operator

OFFSET returns the offset part (low-order word) of the expression following the operator.

```
OFFSET Expression
```

Result

An immediate value.

Inline Assembler SEG Operator

SEG returns the segment part (high-order word) of the expression following the operator.

```
SEG Expression
```

Result

An immediate value.

Inline Assembler TYPE Operator

TYPE returns the type (size in bytes) of the expression following the operator.

```
TYPE Expression
```

The type of an immediate value is 0.

Inline Assembler Typecast Operator (... PTR ...)

PTR casts the second expression to the type of the first expression.

```
Expression1 PTR Expression2
```

Result

A memory reference with the value of the second expression and the type of the first expression.

Inline Assembler Multiplication Operator (... * ...)

The * operator multiplies the first expression by the second expression.

```
Expression1 * Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Integer Division (... / ...)

The / operator divides the first expression by the second expression and returns the integer part of the operation.

```
Expression1 / Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Integer Modulus (... MOD ...)

MOD divides the first expression by the second expression and returns the remainder part of the operation.

```
Expression1 MOD Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Shift Left Operator (... SHL ...)

SHL shifts the first expression to the left by nnn bits, where nnn is the second expression.

```
Expression1 SHL Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Shift Right Operator (... SHR ...)

SHR shifts the first expression to the right by nnn bits, where nnn is the second expression.

```
Expression1 SHR Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Addition Operator (... + ...)

The + operator adds the first expression to the second expression (or vice versa).

```
Expression1 + Expression2
```

The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value.

Result

Relocatable value if one of the expressions is a relocatable value.

Memory reference if either of the expressions is a memory reference.

Inline Assembler Subtraction Operator (... - ...)

The - operator subtracts the second it from the first expression.

```
Expression1 - Expression2
```

The first expression can have any class, but the second expression must be an absolute immediate value.

Result

Has the same class as the first expression.

Inline Assembler Bitwise Negation (NOT)

NOT returns the bitwise negative (1's complement) of the expression after it.

```
NOT Expression
```

The expression must be an absolute immediate value.

Result

Absolute immediate value.

Inline Assembler Bitwise AND

AND returns the bitwise AND of the two expressions.

```
Expression1 AND Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Bitwise OR

OR returns the bitwise OR of the two expressions.

```
Expression1 OR Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Bitwise Exclusive XOR

XOR returns the bitwise exclusive-OR of the two expressions.

```
Expression1 XOR Expression2
```

Both expressions must be absolute immediate values.

Result

Absolute immediate value.

Inline Assembler Function Returns

Functions using the inline assembler directive (asm) must return their results as follows:

Result	Returned In
Ordinal	AL (8-bit values)
	AX (16-bit values)
	DX:AX (32-bit values)
Real	DX:BX:AX
8087	ST(0) on the 8087's register stack
Pointer	DX:AX

short jump

1-byte opcode followed by 1-byte displacement.

near jump

1-byte opcode followed by 2-byte displacement.

short inverse jump

A short jump with the inverse condition jumps over a near jump to the target label (5 bytes in total).

ANSI Implementation-specific standards

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually. This topic describes how Borland has chosen to define these implementation-specific standards. The section numbers refer to the February 1990 ANSI Standard. Remember that there are differences between C and C++; this topic addresses C only.

2.1.1.3 How to identify a diagnostic.

When the compiler runs with the correct combination of options, any messages it issues beginning with the words *Fatal, Error,* or *Warning* are diagnostics in the sense that ANSI specifies. The options needed to ensure this interpretation are as follows:

Table 1 Identifying diagnostics in Borland C++

Option	Action
–A	Enable only ANSI keywords.
–C–	No nested comments allowed.
–i32	At least 32 significant characters in identifiers.
–p–	Use C calling conventions.
–w–	Turn off all warnings except the following.
–wbei	Turn on warning about inappropriate initializers.
–wbig	Turn on warning about constants being too large.
–wcpt	Turn on warning about nonportable pointer comparisons.
–wdcl	Turn on warning about declarations without type or storage class.
–wdup	Turn on warning about duplicate nonidentical macro definitions.
–wext	Turn on warning about variables declared both as external and as static.
–wfdt	Turn on warning about function definitions using a typedef.
–wrpt	Turn on warning about nonportable pointer conversion.
–wstu	Turn on warning about undefined structures.
–wsus	Turn on warning about suspicious pointer conversion.
–wucp	Turn on warning about mixing pointers to signed and unsigned char.
–wvrt	Turn on warning about void functions returning a value.

You cannot use the following options:

–ms!	SS must be the same as DS for small data models.
–mm!	SS must be the same as DS for small data models.
–mt!	SS must be the same as DS for small data models.
–zG*xx*	The BSS group name cannot be changed.
–zS*xx*	The data group name cannot be changed.

Other options not specifically mentioned here can be set to whatever you want.

2.1.2.2.1 The semantics of the arguments to main.

The value of *argv*[0] is a pointer to a null byte when the program is run on DOS versions prior to version 3.0. For DOS version 3.0 or later, *argv*[0] points to the program name.

The remaining *argv* strings point to each component of the DOS command-line arguments. Whitespace separating arguments is removed, and each sequence of contiguous non-whitespace characters is

treated as a single argument. Quoted strings are handled correctly (that is, as one string containing spaces).

2.1.2.3 What constitutes an interactive device.

An interactive device is any device that looks like the console.

2.2.1 The collation sequence of the execution character set.

The collation sequence for the execution character set uses the signed value of the character in ASCII.

2.2.1 Members of the source and execution character sets.

The source and execution character sets are the extended ASCII set supported by the IBM PC. Any character other than Ctrl+Z can appear in string literals, character constants, or comments.

2.2.1.2 Multibyte characters.

Multibyte characters are supported in Borland C++.

2.2.2 The direction of printing.

Printing is from left-to-right, the normal direction for the PC.

2.2.4.2 The number of bits in a character in the execution character set.

There are 8 bits per character in the execution character set.

3.1.2 The number of significant initial characters in identifiers.

The first 32 characters are significant, although you can use a command-line option (–**i**) to change that number. Both internal and external identifiers use the same number of significant characters. (The number of significant characters in C++ identifiers is unlimited.)

3.1.2 Whether case distinctions are significant in external identifiers.

The compiler normally forces the linker to distinguish between uppercase and lowercase. You can use a command-line option (–**l–c**) to suppress the distinction.

3.1.2.5 The representations and sets of values of the various types of integers.

Table 2 Identifying diagnostics in C++

Type	16-bit minimum value	16-bit maximum value	32-bit minimum value	32-bit maximum value
signed char	–128	127	–128	127
unsigned char	0	255	0	255
signed short	–32,768	32,767	–32,768	32,767
unsigned short	0	65,535	0	65,535
signed int	–32,768	32,767	–2,147,483,648	–2,147,483,647
unsigned int	0	65,535	0	4,294,967,295
signed long	–2,147,483,648	2,147,483,647	–2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295	0	4,294,967,295

All **char** types use one 8-bit byte for storage.

All **short** types use 2 bytes, whether in a 16- or 32-bit program.

All **short** and **int** types use 2 bytes (in 16-bit programs).

All **int** types use 4 bytes (in 32-bit programs).

All **long** types use 4 bytes.

If alignment is requested (–**a**), all non**char** integer type objects will be aligned to even byte boundaries. If

the requested alignment is **–a4**, the result is 4-byte alignment. Character types are never aligned.

3.1.2.5 The representations and sets of values of the various types of floating-point numbers.

The IEEE floating-point formats as used by the Intel 8087 are used for all Borland C++ floating-point types. The **float** type uses 32-bit IEEE real format. The **double** type uses 64-bit IEEE real format. The **long double** type uses 80-bit IEEE extended real format.

3.1.3.4 The mapping between source and execution character sets.

Any characters in string literals or character constants remain unchanged in the executing program. The source and execution character sets are the same.

3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.

Wide characters are supported.

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.

Wide character constants are recognized.

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.

Character constants can contain one or two characters. If two characters are included, the first character occupies the low-order byte of the constant, and the second character occupies the high-order byte.

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.

These conversions are performed by simply truncating the high-order bits. Signed integers are stored as two's complement values, so the resulting number is interpreted as such a value. If the high-order bit of the smaller integer is nonzero, the value is interpreted as a negative value; otherwise, it is positive.

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.

The integer value is rounded to the nearest representable value. Thus, for example, the **long** value (231 –1) is converted to the **float** value 231. Ties are broken according to the rules of IEEE standard arithmetic.

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.

The value is rounded to the nearest representable value. Ties are broken according to the rules of IEEE standard arithmetic.

3.3 The results of bitwise operations on signed integers.

The bitwise operators apply to signed integers as if they were their corresponding unsigned types. The sign bit is treated as a normal data bit. The result is then interpreted as a normal two's complement signed integer.

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.

The access is allowed and the different type member will access the bits stored there. You'll need a detailed understanding of the bit encodings of floating-point values to understand how to access a floating-type member using a different member. If the member stored is shorter than the member used to access the value, the excess bits have the value they had before the short member was stored.

3.3.3.4 The type of integer required to hold the maximum size of an array.

For a normal array, the type is **unsigned int**, and for huge arrays the type is **signed long.**

3.3.4 The result of casting a pointer to an integer or vice versa.

When converting between integers and pointers of the same size, no bits are changed. When converting from a longer type to a shorter type, the high-order bits are truncated. When converting from a shorter integer type to a longer pointer type, the integer is first widened to an integer type the same size as the pointer type.

Thus signed integers will sign-extend to fill the new bytes. Similarly, smaller pointer types being converted to larger integer types will first be widened to a pointer type as wide as the integer type.

3.3.5 The sign of the remainder on integer division.

The sign of the remainder is negative when only one of the operands is negative. If neither or both operands are negative, the remainder is positive.

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t.

The type is signed **int** when the pointers are **near** (or the program is a 32-bit application), or **signed long** when the pointers are **far** or **huge**. The type of ptrdiff_t depends on the memory model in use. In small data models, the type is **int**. In large data models, the type is **long**.

3.3.7 The result of a right shift of a negative signed integral type.

A negative signed value is sign extended when right shifted.

3.5.1 The extent to which objects can actually be placed in registers by using the *register* storage-class specifier.

Objects declared with any two-byte integer or pointer types can be placed in registers. The compiler places any small auto objects into registers, but objects explicitly declared as register take precedence. At least two and as many as six registers are available. The number of registers actually used depends on what registers are needed for temporary values in the function.

3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field.

Plain **int** bit fields are treated as **signed int** bit fields.

3.5.2.1 The order of allocation of bit fields within an int.

Bit fields are allocated from the low-order bit position to the high-order.

3.5.2.1 The padding and alignment of members of structures.

By default, no padding is used in structures. If you use the word alignment option (–**a**), structures are padded to even size, and any members that do not have character or character array type are aligned to an even multiple offset.

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.

When alignment (–**a**) is not requested, bit fields can straddle word boundaries, but are never stored in more than two adjacent bytes.

3.5.2.2 The integer type chosen to represent the values of an enumeration type.

Store all enumerators as full **int**s. Store the enumerations in a **long** or **unsigned long** if the values don't fit into an **int**. This is the default behavior as specified by **–b** compiler option.

The **–b**- behavior specifies that enumerations should be stored in the smallest integer type that can represent the values. This includes all integral types, for example, **signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long,** and **unsigned long**.

For C++ compliance, **–b-** must be specified because it is not correct to store all enumerations as **int**s for C++.

3.5.3 What constitutes an access to an object that has volatile-qualified type.

Any reference to a volatile object will access the object. Whether accessing adjacent memory locations will also access an object depends on how the memory is constructed in the hardware. For special device memory, such as video display memory, it depends on how the device is constructed. For normal PC memory, volatile objects are used only for memory that might be accessed by asynchronous interrupts, so accessing adjacent objects has no effect.

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.

There is no specific limit on the number of declarators. The number of declarators allowed is fairly large, but when nested deeply within a set of blocks in a function, the number of declarators will be reduced. The number allowed at file level is at least 50.

3.6.4.2 The maximum number of case values in a switch statement.

There is no specific limit on the number of cases in a switch. As long as there is enough memory to hold the case information, the compiler will accept them.

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.

All character constants, even constants in conditional directives, use the same character set (execution). Single-character character constants will be negative if the character type is signed (default and –**K** not requested).

3.8.2 The method for locating includable source files.

For include file names given with angle brackets, if include directories are given in the command line, then the file is searched for in each of the include directories. Include directories are searched in this order: first, using directories specified on the command line, then using directories specified in TURBOC.CFG or BCC32.CFG. If no include directories are specified, then only the current directory is searched.

3.8.2 The support for quoted names for includable source files.

For quoted file names, the file is first searched for in the current directory. If not found, searches for the file as if it were in angle brackets.

3.8.2 The mapping of source file name character sequences.

Backslashes in include file names are treated as distinct characters, not as escape characters. Case differences are ignored for letters.

3.8.8 The definitions for _ _DATE_ _ and _ _TIME_ _ when they are unavailable.

The date and time are always available and will use the operating system date and time.

4.1.1 The decimal point character.

The decimal point character is a period (.).

4.1.5 The type of the sizeof operator, *size_t*.

The type *size_t is* ***unsigned.***

4.1.5 The null pointer constant to which the macro NULL expands.

For a 16-bit application, an integer or a long 0, depending on the memory model.

For 32-bit applications, NULL expands to an **int** zero or a **long** zero. Both are 32-bit **signed** numbers.

4.2 The diagnostic printed by and the termination behavior of the assert function.

The diagnostic message printed is “Assertion failed: *expression*, file *filename*, line *nn*”, where *expression* is the asserted expression that failed, *filename* is the source file name, and *nn* is the line number where the assertion took place.

Abort is called immediately after the assertion message is displayed.

4.3 The implementation-defined aspects of character testing and case-mapping functions.
None, other than what is mentioned in 4.3.1.

4.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions.
First 128 ASCII characters for the default C locale. Otherwise, all 256 characters.

4.5.1 The values returned by the mathematics functions on domain errors.
An IEEE NAN (not a number).

4.5.1 Whether the mathematics functions set the integer expression *errno* to the value of the macro ERANGE on underflow range errors.
No, only for the other errors—domain, singularity, overflow, and total loss of precision.

4.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.
No; fmod(x,0) returns 0.

4.7.1.1 The set of signals for the signal function.
SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM.

4.7.1.1 The semantics for each signal recognized by the signal function.
See the description of signal.

4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function.
See the description of signal.

4.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed.
The equivalent of signal(sig, SIG_DFL) is always executed.

4.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.
No, it is not.

4.9.2 Whether the last line of a text stream requires a terminating newline character.
No, none is required.

4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.
Yes, they do.

4.9.2 The number of null characters that may be appended to data written to a binary stream.
None.

4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.
The file position indicator of an append-mode stream is initially placed at the beginning of the file. It is reset to the end of the file before each write.

4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point.
A write of 0 bytes might or might not truncate the file, depending on how the file is buffered. It is safest to

classify a zero-length write as having indeterminate behavior.

4.9.3 The characteristics of file buffering.

Files can be fully buffered, line buffered, or unbuffered. If a file is buffered, a default buffer of 512 bytes is created upon opening the file.

4.9.3 Whether a zero-length file actually exists.

Yes, it does.

4.9.3 Whether the same file can be open multiple times.

Yes, it can.

4.9.4.1 The effect of the remove function on an open file.

No special checking for an already open file is performed; the responsibility is left up to the programmer.

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.

Rename returns a –1 and *errno* is set to EEXIST.

4.9.6.1 The output for %p conversion in fprintf.

In near data models, four hex digits (XXXX). In far data models, four hex digits, colon, four hex digits (XXXX:XXXX). (For 16-bit programs.)

Eight hex digits (XXXXXXXX). (For 32-bit programs.)

4.9.6.2 The input for %p conversion in fscanf.

See 4.9.6.1.

4.9.6.2 The interpretation of a –(hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.

See the description of scanf**.**

4.9.9.1 The value the macro errno is set to by the fgetpos or ftell function on failure.

EBADF Bad file number.

4.9.10.4 The messages generated by perror.

Table 3 Messages generated in both Win16 and Win32

Arg list too big	Math argument
Attempted to remove current directory	Memory arena trashed
Bad address	Name too long
Bad file number	No child processes
Block device required	No more files
Broken pipe	No space left on device
Cross-device link	No such device
Error 0	No such device or address
Exec format error	No such file or directory
Executable file in use	No such process
File already exists	Not a directory
File too large	Not enough memory
Illegal seek	Not same device
Inappropriate I/O control operation	Operation not permitted

Input/output error	Path not found
Interrupted function call	Permission denied
Invalid access code	Possible deadlock
Invalid argument	Read-only file system
Invalid data	Resource busy
Invalid environment	Resource temporarily unavailable
Invalid format	Result too large
Invalid function number	Too many links
Invalid memory block address	Too many open files
Is a directory	

Table 4 Messages generated only in Win32

Bad address	No child processes
Block device required	No space left on device
Broken pipe	No such device or address
Executable file in use	No such process
File too large	Not a directory
Illegal seek	Operation not permitted
Inappropriate I/O control operation	Possible deadlock
Input/output error	Read-only file system
Interrupted function call	Resource busy
Is a directory	Resource temporarily unavailable
Name too long	Too many links

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.

calloc and malloc will ignore the request and return 0. realloc will free the block.

4.10.4.1 The behavior of the abort function with regard to open and temporary files.

The file buffers are not flushed and the files are not closed.

4.10.4.3 The status returned by exit if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.

Nothing special. The status is returned exactly as it is passed. The status is a represented as a **signed char.**

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.

The environment strings are those defined in the operating system with the SET command. *putenv* can be used to change the strings for the duration of the current program, but the SET command must be used to change an environment string permanently.

4.10.4.5 The contents and mode of execution of the string by the system function.

The string is interpreted as an operating system command. COMSPEC is used or COMMAND.COM is executed (for 16-bit programs) or CMD.EXE (for 32-bit programs) and the argument string is passed as a command to execute. Any operating system built-in command, as well as batch files and executable programs, can be executed.

4.11.6.2 The contents of the error message strings returned by strerror.

See 4.9.10.4.

4.12.1 The local time zone and Daylight Saving Time.

Defined as local PC time and date.

4.12.2.1 The era for clock.

Represented as clock ticks, with the origin being the beginning of the program execution.

4.12.3.5 The formats for date and time.

Borland C++ implements ANSI formats.

Floating-point I/O

See also

Floating-point output requires linking of conversion routines used by *printf*, *scanf*, and any variants of these functions. To reduce executable size, the floating-point formats are not automatically linked. However, this linkage is done automatically whenever your program uses a mathematical routine or the address is taken of some floating-point number. If neither of these actions occur, the missing floating-point formats can result in a run-time error.

Example

The following program illustrates how to set up your program to properly execute.

```
/* PREPARE TO OUTPUT FLOATING-POINT NUMBERS. */
   #Include <stdio.h>

   #pragma extref _floatconvert

   void main() {
      printf("d = %f\n", 1.3);
      }
```

Floating-point options

See also

There are two types of numbers you work with in C: *integer* (**int**, **short**, **long**, and so on) and *floating point* (**float**, **double**, and **long double**). Your computer's processor can easily handle integer values, but more time and effort are required to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

Note: If you have an 80486 or Pentium processor, the numeric coprocessor is probably already built in.

Emulating the 80x87 chip

See also

The default Borland C++ code-generation option is *emulation* (the **–f** command-line compiler option). This option is for programs that might or might not have floating point, and for machines that might or might not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it uses the 80x87 if it is present; if no coprocessor is present at run time, it uses special software that emulates the 80x87. This software uses 512 bytes of your stack, so make allowance for it when using the emulation option and set your stack size accordingly.

Using the 80x87 code

See also

If your program is going to run only on machines that have an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Choose the 80x87 floating-point code-generation option (the **–f87** command-line compiler option). Will then link your programs with FP87.LIB instead of with EMU.LIB.

No floating-point code

See also

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code-generation option (the **–f–** command-line compiler option). Then will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

Fast floating-point option

See also

Borland C++ has a fast floating-point option (the **–ff** command-line compiler option). It can be turned off with **–ff–** on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;
x = (float)(3.5*x);
```

To execute this correctly, *x* is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in *x*. Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

The 87 environment variable

See also

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

Why to override the default autodetection

There are some situations in which you might want to override this default autodetection behavior. For example, your own run-time system might have an 80x87, but you might need to verify that your program will work as intended on systems without a coprocessor. Or your program might need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

How to override the default autodetection

Borland C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

Defining the 87 environment variable

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces on either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.

Note: Setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. *Let the programmer beware:* If you set 87 = Y when, in fact, there is no 80x87 available on that system, your system will hang.

Undefining the 87 environment variable

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press Enter immediately after typing the equal sign.

Registers and the 80x87

See also

When you use floating point, make note of these points about registers:

- In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
- If you are mixing floating point with inline assembly, you might need to take special care when using 80x87 registers. Unless you are sure that enough free registers exist, you might need to save and pop the 80x87 registers before calling functions that use the coprocessor.

Disabling floating-point exceptions

See also

By default, programs abort if a floating-point overflow or divide-by-zero error occurs. You can mask these floating-point exceptions by a call to *_control87* in *main*, before any floating-point operations are performed.

Example

```
#include <float.h>
main() {
   _control87(MCW_EM,MCW_EM);
   .
   .
   .
}
```

You can determine whether a floating-point exception occurred after the fact by calling _status87 or _clear87.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN is likely to cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of *_matherr* into your program:

```
#include <math.h>
int _matherr(struct _exception *e)
{
   return 1;              /* error has been handled */
}
```

Any other use of *_matherr* to intercept math errors is not encouraged; it is considered obsolete and might not be supported in future versions of Borland C++.

Running out of memory

See also

Borland C++ does not generate any intermediate data structures to disk when it is compiling (writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message “Out of memory” if there isn’t enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions.

Memory models

See also

Borland C++ gives you six memory models, each suited for different program and code sizes. Each memory model uses memory differently. What do you need to know to use memory models?

To answer that question, you need to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; an 80286, 80386, 80486, or Pentium. For now, we'll just refer to it as an 8086.

The 8086 registers

See also

This table shows some of the registers found in the 8086 processor. There are other registers—because they can't be accessed directly, they aren't shown here.

General-purpose registers

AX	accumulator (math operations)	
	AH	AL
BX	base (indexing)	
	BH	BL
CX	count (indexing)	
	CH	CL
DX	data (holding data)	
	DH	DL

Segment address registers

CS	code segment pointer
DS	data segment pointer
SS	stack segment pointer
ES	extra segment pointer

Special-purpose registers

SP	stack pointer
BP	base pointer
SI	source index
DI	destination index

General-purpose registers

See also

The general-purpose registers are the registers used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.
- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers

See also

The segment registers hold the starting address of each of the four segments. As described in Address calculation, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers

See also

The 8086 also has some special-purpose registers:

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

Borland C++ functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP points to the saved previous BP value if there is a stack frame. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off.*

Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to local variables.

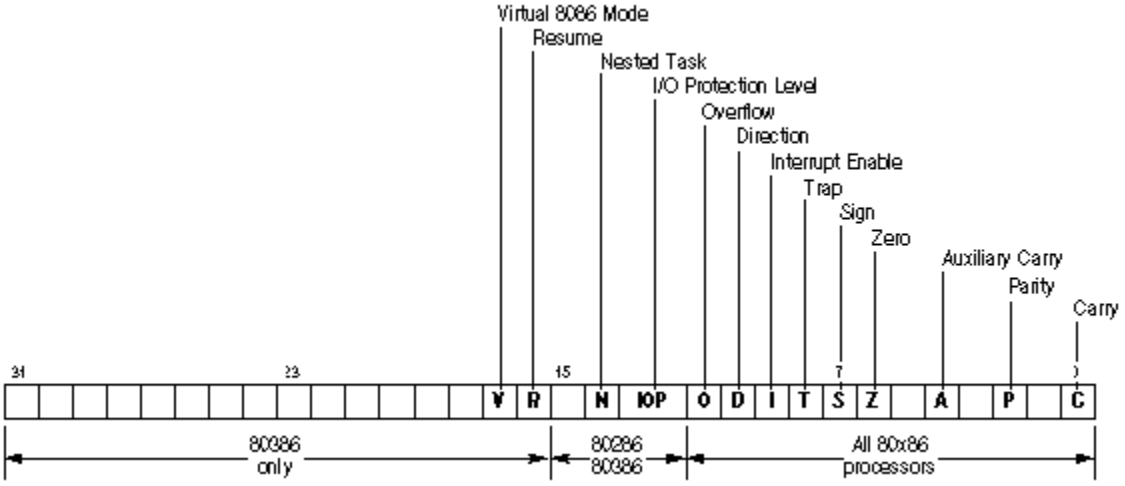
The flags register

See also

The 16-bit flags register contains all pertinent information about the state of the 8086 and the results of recent instructions.

for example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the *carry* and *overflow flags,* similarly report the results of arithmetic and logical operations.

Flags register of the 80x86 processors



Other flags control the 8086 operation modes. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD, STI,** and **CMC)** and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ, RCR,** and **MOVSB.** The flags register is not really used as a storage location, but rather holds the status and control data for the 8086.

Memory segmentation

See also

The Intel 8086 microprocessor has a *segmented memory architecture*. It has a total address space of 1 MB, but is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase “segmented memory architecture."

- The 8086 keeps track of four different segments: code, data, stack, and extra. The code segment is where the machine instructions are; the data segment is where information is; the stack is, of course, the stack; and the extra segment is also used for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the *code, data, stack,* and *extra* segments, respectively.
- A segment can be located anywhere in memory. In DOS real mode it can be located almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that’s evenly divisible by 16 (in decimal).

Address calculation

See also

Note: This discussion is applicable only to real mode under DOS. You can safely ignore it for Windows development.

A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment: how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

```
DS register (shifted):  0010 1111 1000 0100 0000  =  2F840
Offset:                      0000 0101 0011 0010  =  00532
_____________________   ____________________________________

Address:                0010 1111 1101 0111 0010  =  2FD72
```

Note: A chunk of 16 bytes is known as a *paragraph,* so you could say that a segment always starts on a paragraph boundary.

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

the standard notation for an address takes the form *segment:offset;* for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

Segments can overlap (but don’t have to). For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that’s all the space you’d have for your code, your data, and your stack.

Pointers

See also

Although you can declare a pointer or function to be a specific type regardless of the model used, by default the type of memory model you choose determines the default type of pointers used for code and data. There are four types of pointers: near (16 bits), far (32 bits), huge (also 32 bits), and segment (16 bits).

Near pointers

See also

A near pointer (16-bits) relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far pointers

See also

A far pointer (32-bits) contains not only the offset within the segment, but also the segment address (as another 16-bit value), which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; this, in turn, allows you to have programs larger than 64K. You can also address more than 64K of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in Address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a, b,* and *c*—containing those three values respectively, then all the following expressions would be *false:*

```
if (a == b) . . .
if (b == c) . . .
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the **>, >=, <,** and **<=** operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a, b*, and *c* still have the values previously listed, the following expressions would all be *true:*

```
if (a > b) . . .
if (b > c) . . .
if (a > c) . . .
```

The equals (**==**) and not-equal (**!=**) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (**<=, >=, <,** and **>**) use just the offset.

The **==** and **!=** operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

Note: If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge pointers

See also

Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, they are *normalized* to avoid the problems associated with far pointers.

A normalized pointer is a 32-bit pointer that has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

There are three reasons why it is important to always keep huge pointers normalized:

1. For any given memory address there is only one possible huge address (segment:offset) pair. That means that the **==** and **!=** operators return correct answers for any huge pointers.
2. in addition, the **>, >=, <,** and **<=** operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results of these comparisons will also be correct.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **struct**s that's larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

The six memory models

See also

Borland C++ gives you six memory models for 16-bit DOS programs: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

- **Tiny**. As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the **/t** option. Use this model when memory is at an absolute premium.
- **Small**. The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used. This is a good size for average applications.
- **Medium**. Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MB. This model is best for large programs without much data in memory.
- **Compact**. The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1 MB range. This model is best if code is small but needs to address a lot of data.
- **Large**. Far pointers are used for both code and data, giving both a 1 MB range. Large and huge are needed only for very large applications.
- **Huge**. Far pointers are used for both code and data. Normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

The following figures show how memory in the 8086 is apportioned for the memory models. To select these memory models, you can either use menu selections from the IDE or you can type options invoking the command-line compiler version of .

Tiny model memory segmentation

Small model memory segmentation

Medium model memory segmentation

Compact model memory segmentation

Large model memory segmentation

Huge model memory segmentation

Comparison of memory models

The following table summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are small (64K) or large (16 MB); these groups correspond to the rows and columns in the table.

Data size	Code size = 64K	Code size = 16MB
64K	Tiny (data, code overlap; total size = 64K)	
	Small (no overlap; total size = 128K)	Medium (small data, large code)
16 MB	Compact (large data, small code)	Large (large data, code)
		Huge (same as large but static data > 64K)

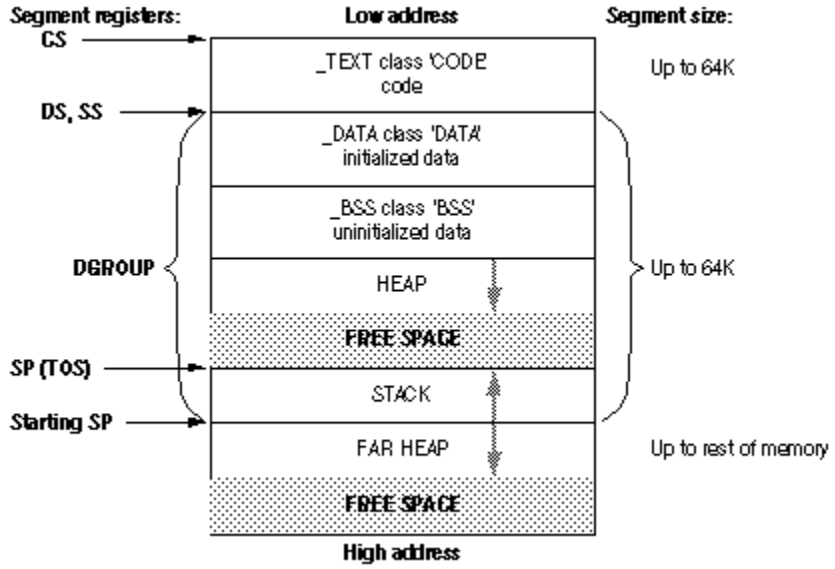
The models tiny, small, and compact are small code models because, by default, code pointers are

near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

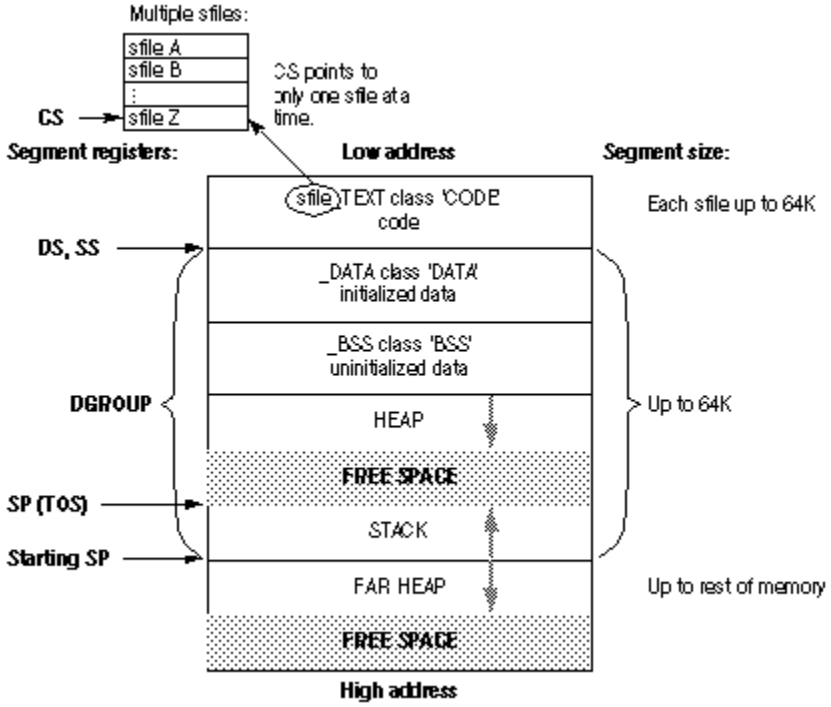
When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code models (medium, large, or huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in each module.

Tiny model memory segmentation

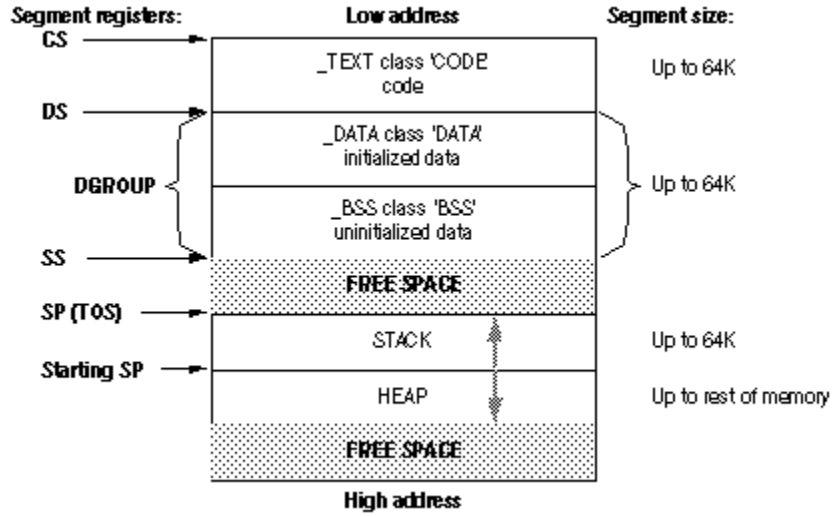
Small model memory segmentation



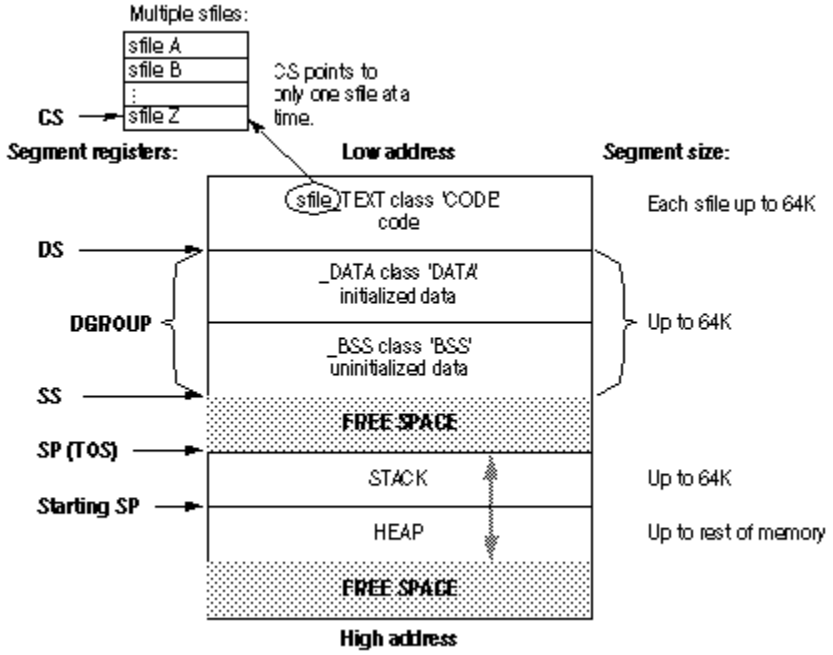
Medium model memory segmentation



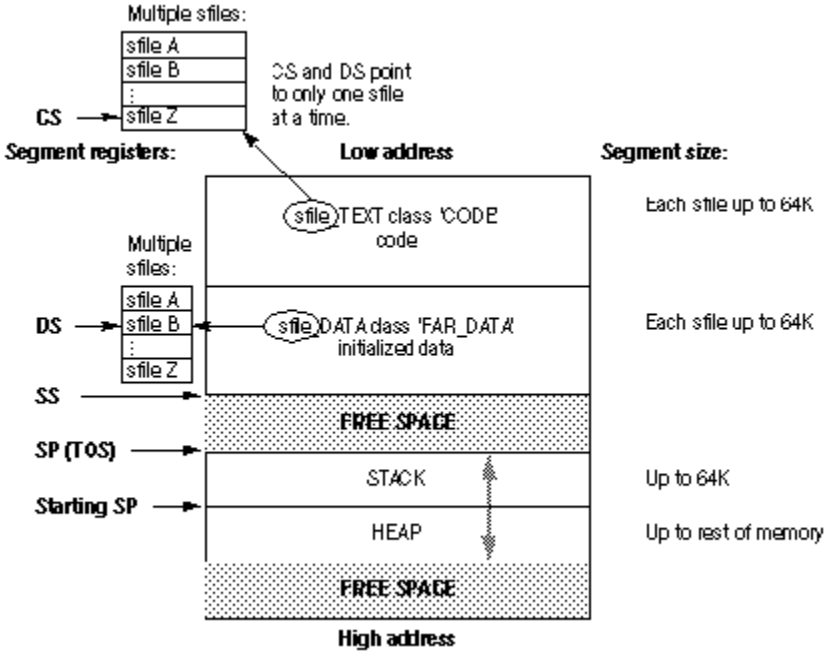
Compact model memory segmentation



Large model memory segmentation



Huge model memory segmentation



Mixed-model programming: Addressing modifiers

See also

Borland C++ introduces eight new keywords not found in standard ANSI C. These keywords are **_ _near, _ _far, _ _huge, _ _cs, _ _ds, _ _es, _ _ss,** and **_ _seg.** These keywords can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Borland C++ , you can modify the declarations of pointers, objects, and functions with the keywords **_ _near, _ _far,** or **_ _huge.** (See Pointers for more information on the **_ _near, _ _far,** and **_ _huge** data pointers.) You can declare far objects using the **_ _far** keyword. **_ _Near** functions are invoked with near calls and exit with near returns. Similarly, **_ _far** functions are called **_ _far** and return far values. **_ _huge** functions are like **_ _far** functions, except that **_ _huge** functions set DS to a new value, and **_ _far** functions do not.

There are also four special **_ _near** data pointers: **_ _cs, _ _ds, _ _es,** and **_ _ss.** These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

Then *p* would contain a 16-bit offset into the stack segment.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The following table shows how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 MB memory space (far, has its own segment address).

Memory model	Function pointers	Data pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Segment pointers

See also

Use **_ _seg** in segment pointer type declarators. The resulting pointers are 16-bit segment pointers. The syntax for **_ _seg** is:

datatype _seg **identifier;*

For example,

```
int _seg *name;
```

Any indirection through *identifier* has an assumed offset of 0. In arithmetic involving segment pointers the following rules hold true:

1. You can't use the **++, - -, +=,** or **-=** operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. When adding a near pointer to a segment pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer. Therefore, the two pointers must either point to the same type, or one must be a pointer to void. There is no multiplication of the offset regardless of the type pointed to.
4. When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
5. When adding or subtracting an integer operand to or from a segment pointer, the result is a far pointer, with the segment taken from the segment pointer and the offset found by multiplying the size of the object pointed to by the integer operand. The arithmetic is performed as if the integer were added to or subtracted from the far pointer.
6. Segment pointers can be assigned, initialized, passed into and out of functions, compared and so forth. (Segment pointers are compared as if their values were **unsigned** integers.) in other words, other than the above restrictions, they are treated exactly like any other pointer.

Declaring far objects

See also

You can declare far objects in Borland C++. For example,

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

The command-line compiler options **–zE, –zF,** and **–zH** (which can also be set using **#pragma option**) affect the far segment name, class, and group, respectively. When you use **#pragma option**, you can make them apply to any ensuing far object declarations. Thus you could use the following sequence to create a far object in a specific segment:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH* -zF*
```

This will put *x* in segment MYSEGMENT 'MYCLASS' in the group 'MYGROUP', then reset all of the far object items to the default values. Note that by using these options, several far objects can be forced into a single segment:

```
#pragma option -zEcombined -zFmyclass
int far x;
double far y;
#pragma option -zE* -zF*
```

Both *x* and *y* will appear in the segment COMBINED 'MYCLASS' with no group.

Declaring functions to be near or far

See also

On occasion, you'll want (or need) to override the default function type of your memory model.

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp) {
   if (exp <= 0)
      return(1);
   else
      return(x * power(x, exp-1));
   }
```

Every time *power* calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring power as **_ _near,** you eliminate some of the overhead by forcing all calls to that function to be near:

```
double _ _near power(double x,int exp)
```

This guarantees that *power* is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you're using a large code model (medium, large, or huge), you can only call *power* from within the module where it is defined. Other modules have their own code segment and thus cannot call **_ _near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the power example at the beginning of this topic. It is wise to also declare power as static, since it should be called only from within the current module. That way, being a static, its name will not be available to any functions outside the module.

Declaring pointers to be near, far, or huge

See also

You've seen why you might want to declare functions to be of a different model than the rest of the program. For the same reasons given in Declaring functions to be near or far, you might want to modify pointer declarations: either to avoid unnecessary overhead (declaring _ _near when the default would be _ _far) or to reference something outside of the default segment (declaring _ _far or _ _huge when the default would be _ _near).

There are, of course, potential pitfalls in declaring functions and pointers to be of nondefault types. For example, say you have the following small model program:

```
void myputs(s) {
   char *s;
   int  i;
   for (i = 0; s[i] != 0; i++) putc(s[i]);
   }
main() {
   char  near *mystr;
mystr = "Hello, world\n"
     myputs(mystr);
   }
```

This program works fine. In fact, the **_ _near** declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in *main* is still near (it's still a 16-bit pointer). However, the pointer *s* in *myputs* is now far, because that's the default. This means that *myputs* will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr.*

How do you avoid this problem? If you're going to explicitly declare pointers to be of type **_ _far** or **_ _near,** be sure to use function prototypes for any functions that might use them. The solution is to define ***myputs*** in ANSI C style, like this:

```
void myputs(char *s) {
   /* body of myputs */
   }
```

Now when Borland C++ compiles your program, it knows that *myputs* expects a pointer to **char**; and since you're compiling under the large model, it knows that the pointer must be **_ _far**. Because of that, will push the data segment (DS) register onto the stack along with the 16-bit value of ***mystr***, forming a far pointer.

How about the reverse case: arguments to *myputs* declared as ***_ _far*** and compiled with a small data model? Again, without the function prototype, you will have problems, because *main* will push both the offset and the segment address onto the stack, but *myputs* will expect only the offset. With the prototype-style function definitions, though, *main* will only push the offset onto the stack.

Pointing to a given segment:offset address

See also

You can make a far pointer point to a given memory location (a specific segment:offset address). You can do this with the macro *MK_FP,* which takes a segment and an offset and returns a far pointer. For example,

```
MK_FP(segment_value, offset_value)
```

Given a **_ _far** pointer, **fp**, you can get the segment component with *FP_SEG(fp)* and the offset component with *FP_OFF(fp)*. For more information about these three Borland C++ library routines, refer to the Run-time Library Reference.

Using library files

See also

Borland C++ offers a version of the standard library routines for each of the six memory models. Is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Borland C++ linker, TLINK, directly (as a standalone linker), you need to specify which libraries to use. See Using TLINK and TLINK32 for details on how to do this.

Linking mixed modules

See also

Suppose you compiled one module using the small memory model and another module using the large model, then wanted to link them together. This would present some problems, but they can be solved.

The files would link together fine, but the problems you would encounter would be similar to those described in the topic, Declaring functions to be near or far. If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as described in the topic, Declaring pointers to be near, far, or huge, since a function in the small module would expect to pass and receive **_ _near** pointers, and a function in the large module would expect **_ _far** pointers.

Using function prototypes

The solution, again, is to use function prototypes. Suppose that you put *myputs* into its own module and compile it with the large memory model. Then create a header file called myputs.h (or some other name with a .h extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, put *main* into its own module (called MYMAIN.C), and set things up like this:

```
#include <stdio.h>
#include "myputs.h"
main() {
   char  near *mystr;
   mystr = "Hello, world\n";
     myputs(mystr);
   }
```

When you compile this program, Borland C++ reads in the function prototype from myputs.h and sees that it is a **_ _far** function that expects a **_ _far** pointer. Therefore, it generates the proper calling code, even if it's compiled using the small memory model.

Linking in library routines

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **_ _far**. To do this, make a copy of each header file you would normally include (such as stdio.h), and rename the copy to something appropriate (such as fstdio.h).

Then edit each function prototype in the copy so that it is explicitly **_ _far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **_ _far** calls be made to the routines, but the pointers passed will also be **_ _far** pointers. Modify your program so that it includes the new header file:

```
#include <fstdio.h>
void main() {
   char  near *mystr;
   mystr = "Hello, world\n";
     printf(mystr);
}
```

Compile your program with the command-line compiler BCC then link it with TLINK, specifying a large model library, such as CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Borland C++ Library Routines, by Category

See also Overview

If you know the name of the function you want Help on, see:

Borland C++ Library Routines, by Name

Otherwise, if you do not know the name of a particular function, but you know what type of action it performs, choose one of the following categories:

Classification Routines
Console I/O Routines
Conversion Routines
Diagnostic Routines
Directory Control Routines
EasyWin Routines
Inline Routines
Input/output Routines
Interface Routines
International API Routines (16-bit)
International API Routines (32-bit)
Manipulation Routines
Math Routines
Memory Routines
Miscellaneous Routines
Obsolete Functions
Process Control Routines
Time and Date Routines
Variable Argument List Routines

Borland C++ has several hundred classes, functions, and macros that you call from within your C and C++ programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more. These classes, functions, and macros are collectively referred to as library routines.

Reasons To Access the Run-time Library Source Code

See also

Here are some reasons why you might want to obtain the source code for run-time library routines:

- To write a function similar to, but not the same as, a Borland C++ function. With access to the run-time library source code, you can tailor the library function to suit your needs, and avoid having to write a separate function of your own.
- To know more about the internals of a library function when you debug your code.
- To delete the leading underscores on C symbols.
- To learn programming techniques by studying tight, professionally written library source code.

Because Borland believes strongly in the concept of open architecture, the Borland C++ run-time library source code is available for licensing. Just fill out the order form distributed with your Borland C++ package, include your payment, and Borland will ship you the Borland C++ run-time library source code.

Guidelines for Selecting Run-Time Libraries

See also Overview

Use the following guidelines when selecting which run-time libraries to use:

- 16-bit DLLs are supported only in the large memory model.
- For 32-bit programs, only the flat memory model is supported.
- 32-bit console and GUI programs require different startup code.
- Multithread applications are supported only in 32-bit programs.

Run-time Libraries Overview

Borland C++ has several hundred classes, functions, and macros that you call from within your C and C++ programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more. These classes, functions, and macros are collectively referred to as library routines.

The Borland C++ run-time libraries are divided into static (OBJ and LIB) and dynamic-link (DLL) versions.

- Static libraries are located in the LIB subdirectory of your installation.
- Dynamic-link libraries are located in the BIN subdirectory of your installation.

Several versions of the run-time libraries are available. For example, there are specific versions for each memory-model, debugging, and 16- and 32-bit versions. There are also optional libraries to provide mathematics, containers, ObjectWindows development, and international applications.

Static Run-time Libraries

See also Legend Overview

Listed below are each of the Borland C++ static library names, the operating environment in which it is available, and its use.

File name	Environment	Use
Directory of BC5\LIB		
BIDSI.LIB	Win 16	16-bit dynamic BIDS import library for BIDS50.DLL
BIDSF.LIB	Win32s, Win32	32-bit BIDS library
BIDSFI.LIB	Win32s, Win32	32-bit dynamic BIDS import library for BIDS50F.DLL
BIDS?.LIB	Win 16	16-bit BIDS library
BWCC.LIB	Win 16	16-bit import library for BWCC.DLL
BWCC32.LIB	Win32s, Win32	32-bit import library for BWCC32.DLL
C0D32.OBJ	Win32s, Win32	32-bit DLL startup module
C0D?.OBJ	Win 16	16-bit DLL startup module
C0W32.OBJ	Win32s, Win32	32-bit GUI EXE startup module
C0W?.OBJ	Win 16	16-bit EXE startup module
C0X32.OBJ	Win32	32-bit console-mode EXE startup module
CRTLDLL.LIB	Win 16	16-bit dynamic import library for BC520RTL.DLL
CT.LIB	DOS	tiny library (DOS only)
CW32.LIB	Win32s, Win32	32-bit GUI single-thread library
CW?.LIB	Win 16	16-bit library
CW32I.LIB	Win32s, Win32	32-bit single-thread, GUI, dynamic RTL import library for CW3230.DLL
CW32MT.LIB	Win32	32-bit GUI multithread library
CW32MTI.LIB	Win32	32-bit multithread, GUI, dynamic RTL import library for CW3220MT.DLL
IMPORT.LIB	Win 16	16-bit import library
IMPORT32.LIB	Win32	Import library; includes Winsock 1.x
INET.LIB	Win32	Import library for the Internet API (URLMON, WININET, HLINK, MSCONF, WEBPOST)
MSEXTRA.LIB	Win32	Import library for some APIs whose module names differ between Win NT and Win 95.
MSWSOCK.LIB	Win32	Import library for MSWSOCK.DLL.
IMPORT32.LIB	Win32s, Win32	32-bit import library
MATHW?.LIB	Win 16	16-bit math libraries
NOEH?.LIB	16-bit DOS, DPMI16	Eliminate exception handling code in run-time libraries
NOEHW?.LIB	Win 16, DPMI16	Eliminate exception handling code in run-time libraries
W32SUT16.LIB	Win 16	16-bit universal thunking library
W32SUT32.LIB	Win32s	32-bit universal thunking library
OBSOLETE.LIB	Win 16, Win32, Win32s	Provides obsolete global variables.
OLE2W16.LIB	Win 16	Import library for the 16-bit OLE 2.0 API
OLE2W32.LIB	Win32	Import library for the 32-bit OLE 2.0 API
RPCEXTRA.LIB	Win32	Import library for some RPC APIs whose names differ between Win NT and Win 95.
TH32.LIB	Win32	Import library for the 32-bit ToolHelp API under Win 95.
W32SUT16.LIB	Win 16	16-bit universal thunking library.

W32SUT32.LIB	Win 32	32-bit universal thunking library.
WS2_32.LIB	Win32	Import library for the 32-bit WinSock 2.0 API.

Directory of BC5\LIB\16BIT

FILES.C	Win 16	Increases the number of file handles
FILES2.C	Win 16	Increases the number of file handles
MATHERR.C	Win 16	Sample of a user-defined floating-point math exception handler for float and double types
MATHERRL.C	Win 16	Sample of a user-defined floating-point math exception handler for long double type
WILDARG.OBJ	Win 16	Transforms wild-card arguments into an array of arguments to *main()* in console-mode applications

Directory of BC5\LIB\32BIT

FILES.C	Win32s, Win32	Increases the number of file handles
FILES2.C	Win32s, Win32	Increases the number of file handles
FILEINFO.OBJ	Win32s, Win32	Passes open file-handle information to child processes
GP.OBJ	Win32s, Win32	Prints register-dump information when an exception occurs
MATHERR.C	Win32s, Win32	Sample of a user-defined floating-point math exception handler for float and double types
MATHERRL.C	Win32s, Win32	Sample of a user-defined floating-point math exception handler for long double type
WILDARGS.OBJ	Win32	Transforms wild-card arguments into an array of arguments to *main()* in console-mode applications

Directory of BC5\LIB\STARTUP

BUILD-C0.BAT	Win 16	Batch file to build C0D?.OBJ, C0F?.OBJ, and C0W?.OBJ
C0.ASM	DOS	Source for C0?.OBJ
C0D.ASM	Win 16	Source for C0D?.OBJ
C0W.ASM	Win 16	Source for C0W?.OBJ
RULES.ASI	Win 16	Assembly rules for C0D.ASM and C0W.ASM

Legend

Each memory model has its own library file and math file that contain versions of the routines written for that particular model.

The **?** placeholder in each of the library file names represents one of the supported memory models (S = small, M = Medium, C = compact, and L = large).

For example, the available versions of the 16-bit DLL startup module (C0D**?**.OBJ) are:

- C0DS.OBJ (small)
- C0DM.OBJ (medium)
- C0DC.OBJ (compact)
- C0DL.OBJ (large)

Dynamic-link Libraries

See also Overview

The dynamic-link library (DLL) version of the run-time library is contained in the BIN subdirectory of your installation. Several versions of the dynamic-link libraries are available. For example, there are 16- and 32-bit specific versions, and versions that support multithread applications.

In the 16-bit specific version, only the large-memory model DLL is provided. No other memory-model is supported in a 16-bit DLL.

Listed below are each of the Borland C++ DLL names, the operating environment in which it is available, and its use.

Directory: BC5\BIN

File Name	Environment	Use
BC520RTL.DLL	Win 16	16-bit, large-memory model
BIDS50.DLL	Win 16	16-bit BIDS
BIDS50F.DLL	Win32s, Win32	32-bit BIDS
CW3230.DLL	Win32s, Win32	32-bit, single thread, GUI mode
CW3220MT.DLL	Win32	32-bit, multithread, GUI mode
LOCALE.BLL	Win 16	Locale library

Default Run-Time Libraries

See also Overview

The following table identifies the default run-time libraries used with each compiler.

Compiler	Environment	Default Libraries
BCC.EXE	16 bit Windows	C0WS.OBJ, CWS.LIB, MATHWS.LIB, IMPORT.LIB
BCC32.EXE	Win32 and Win32s	C0W32.OBJ, CW32.LIB, IMPRTW32.LIB
BCW.EXE	16 bit Windows	Same as BCC.EXE
BCW32.EXE	Win32 and Win32s	Same as BCC32.EXE

C++ Prototyped Routines

See also

Certain routines described in this book have multiple declarations. You must choose the prototype appropriate for your program. In general, the multiple prototypes are required to support the original C implementation and the stricter and sometimes different C++ function declaration syntax. For example, some string-handling routines have multiple prototypes because in addition to the ANSI-C specified prototype, Borland C++ provides prototypes consistent with the ANSI C++ draft.

Function	Header
getvect	dos.h
max	stdlib.h
memchr	string.h
min	stdlib.h
setvect	dos.h
strchr	string.h
strpbrk	string.h
strrchr	string.h
strstr	string.h

Classification Routines

See also

The following routines classify ASCII characters as letters, control characters, punctuation, uppercase, and so.

These routines are all declared in ctype.h.

isalnum	islower
isalpha	isprint
isascii	ispunct
iscntrl	isspace
isdigit	isupper
isgraph	isxdigit

Console I/O Routines

See also

The following routines output text to the screen or read from the keyboard. They cannot be used in a GUI application.

Function	Header	Function	Header
cgets	conio.h	movetext	conio.h
clreol	conio.h	normvideo	conio.h
clrscr	conio.h	putch	conio.h
cprintf	conio.h	puttext	conio.h
cputs	conio.h	_setcursortype	conio.h
delline	conio.h	textattr	conio.h
getpass	conio.h	textbackground	conio.h
gettext	conio.h	textcolor	conio.h
gettextinfo	conio.h	textmode	conio.h
gotoxy	conio.h	ungetc	stdio.h
highvideo	conio.h	wherex	conio.h
insline	conio.h	wherey	conio.h
lowvideo	conio.h	window	conio.h

Conversion Routines

See also

The following routines convert characters and strings from

- alpha to different numeric representations (floating-point, integers, longs)
- numeric to alpha representations
- uppercase to lowercase (and vice versa).

Function	Header	Function	Header
atof	stdlib.h	strtol	stdlib.h
atoi	stdlib.h	_strtold	stdlib.h
atol	stdlib.h	strtoul	stdlib.h
ecvt	stdlib.h	toascii	ctype.h
fcvt	stdlib.h	_tolower	ctype.h
gcvt	stdlib.h	tolower	ctype.h
itoa	stdlib.h	_toupper	ctype.h
ltoa	stdlib.h	toupper	ctype.h
strtod	stdlib.h	ultoa	stdlib.h

Diagnostic Routines

See also

The following routines provide built-in troubleshooting capability.

Function	Header
assert	assert.h
_matherr	math.h
_matherrl	math.h
perror	errno.h

Directory Control Routines

See also

The following routines manipulate directories and path names.

Function	Header	Function	Header
chdir	dir.h	_getdcwd	direct.h
_chdrive	direct.h	getdisk	dir.h
closedir	dirent.h	_makepath	stdlib.h
_dos_findfirst	dos.h	mkdir	dir.h
_dos_findnext	dos.h	mktemp	dir.h
_dos_getdiskfree	dos.h	opendir	direct.h
_dos_getdrive	dos.h	readdir	dirent.h
_dos_setdrive	dos.h	rewinddir	dirent.h
findfirst	dir.h	rmdir	dir.h
findnext	dir.h	_searchenv	stdlib.h
fnmerge	dir.h	searchpath	dir.h
fnsplit	dir.h	_searchstr	stdlib.h
_fullpath	stdlib.h	setdisk	dir.h
getcurdir	dir.h	_splitpath	stdlib.h
getcwd	dir.h		

EasyWin Routines

See also

The following routines are portable to EasyWin programs, but are not available in Windows 16-bit programs. They are provided to help you port DOS programs into a Windows 16-bit applications.

Function	Header	Function	Header
clreol	conio.h	printf	stdio.h
clrscr	conio.h	putch	conio.h
fgetchar	stdio.h	putchar	stdio.h
getch	stdio.h	puts	stdio.h
getchar	stdio.h	scanf	stdio.h
getche	stdio.h	vprintf	stdio.h
gets	stdio.h	vscanf	stdio.h
gotoxy	conio.h	wherex	conio.h
kbhit	conio.h	wherey	conio.h
perror	errno.h		

Inline Routines

See also

The following routines have inline versions. The compiler will generate code for the inline versions when you use #pragma intrinsic or if you specify program optimization..

Function	Header	Function	Header
abs	math.h	stpcpy	string.h
alloca	malloc.h	strcat	string.h
_crotl	stdlib.h	strchr	string.h
_crotr	stdlib.h	strcmp	string.h
_lrotl	stdlib.h	strcpy	string.h
_lrotr	stdlib.h	strlen	string.h
memchr	mem.h	strncat	string.h
memcmp	mem.h	strncmp	string.h
memcpy	mem.h	strncpy	string.h
memset	mem.h	strnset	string.h
_rotl	stdlib.h	strrchr	string.h
_rotr	stdlib.h	strset	string.h

Input/output Routines

See also

The following routines provide stream- and operating-system level I/O capability.

Function	Header	Function	Header
access	io.h	getftime	io.h
chmod	io.h	gets	stdio.h
chsize	io.h	getw	stdio.h
clearerr	stdio.h	ioctl	io.h
close	io.h	isatty	io.h
creat	io.h	kbhit	conio.h
creatnew	io.h	lock	io.h
creattemp	io.h	locking	io.h
cscanf	conio.h	lseek	io.h
_dos_close	dos.h	open	io.h
_pclose	stdio.h	_open_osfhandle	io.h
_dos_creat	dos.h	perror	stdio.h
_dos_creatnew	dos.h	_pipe	io.h
_dos_getfileattr	dos.h	_popen	stdio.h
_dos_getftime	dos.h	printf	stdio.h
_dos_open	dos.h	putc	stdio.h
_dos_read	dos.h	putchar	stdio.h
_dos_setfileattr	dos.h	puts	stdio.h
_dos_setftime	dos.h	putw	stdio.h
_dos_write	dos.h	read	io.h
dup	io.h	remove	stdio.h
dup2	io.h	rename	stdio.h
eof	io.h	rewind	stdio.h
fclose	stdio.h	rmtmp	stdio.h
fcloseall	stdio.h	_rtl_chmod	io.h
fdopen	stdio.h	_rtl_close	io.h
feof	stdio.h	_rtl_creat	io.h
ferror	stdio.h	_rtl_open	io.h
fflush	stdio.h	_rtl_read	io.h
fgetc	stdio.h	_rtl_write	io.h
fgetchar	stdio.h	scanf	stdio.h
fgetpos	stdio.h	setbuf	stdio.h
fgets	stdio.h	setftime	io.h
filelength	io.h	setmode	io.h
fileno	stdio.h	setvbuf	stdio.h
flushall	stdio.h	sopen	io.h

fopen	stdio.h	sprintf	stdio.h
fprintf	stdio.h	sscanf	stdio.h
fputc	stdio.h	strerror	stdio.h
fputchar	stdio.h	_strerror	string.h, stdio.h
fputs	stdio.h	tell	io.h
fread	stdio.h	tempnam	stdio.h
freopen	stdio.h	TFile (class)	file.h
fscanf	stdio.h	tmpfile	stdio.h
fseek	stdio.h	tmpnam	stdio.h
fsetpos	stdio.h	umask	io.h
_fsopen	stdio.h	unlink	dos.h
fstat	sys\stat.h	unlock	io.h
ftell	stdio.h	utime	utime.h
fwrite	stdio.h	vfprintf	stdio.h
get_osfhandle	io.h	vfscanf	stdio.h
getc	stdio.h	vprintf	stdio.h
getch	conio.h	vscanf	stdio.h
getchar	stdio.h	vsprintf	stdio.h
getche	conio.h	vsscanf	io.h

Interface Routines (DOS, 8086, BIOS)

See also

The following routines provide operating system, BIOS and machine-specific capabilities.

Function	Header	Function	Header
bdos	dos.h	inp	conio.h
bdosptr	dos.h	inpw	conio.h
_bios_equiplist	bios.h	inport	dos.h
biosequip	bios.h	inportb	dos.h
biosmemory	bios.h	int86	dos.h
biostime	bios.h	int86x	dos.h
_chain_intr	dos.h	intdos	dos.h
country	dos.h	intdosx	dos.h
ctrlbrk	dos.h	intr	dos.h
_disable	dos.h	MK_FP	dos.h
disable	dos.h	outp	conio.h
dosexterr	dos.h	outpw	conio.h
_dos_getvect	dos.h	outport	dos.h
_dos_setvect	dos.h	outportb	dos.h
_enable	dos.h	parsfnm	dos.h
enable	dos.h	peek	dos.h
FP_OFF	dos.h	peekb	dos.h
FP_SEG	dos.h	poke	dos.h
geninterrupt	dos.h	pokeb	dos.h
getcbrk	dos.h	segread	dos.h
getdfree	dos.h	setcbrk	dos.h
getdta	dos.h	_setcursortype	conio.h
getfat	dos.h	setdta	dos.h
getfatd	dos.h	setvect	dos.h
getpsp	dos.h	setverify	dos.h
getvect	dos.h	sleep	dos.h
getverify	dos.h		

International API Routines (16-bit RTL)

See also

The following routines are affected by the current locale. The current locale is specified by the setlocale function and is enabled by defining __USELOCALES__ with -D command line option. When you define __USELOCALES__, only function versions of the following routines are used in the run-time library rather than macros.

Function	Header	Function	Header
cprintf	stdio.h	scanf	stdio.h
cscanf	stdio.h	setlocale	locale.h
fprintf	stdio.h	sprintf	stdio.h
fscanf	stdio.h	sscanf	stdio.h
isalnum	ctype.h	strcoll	string.h
isalpha	ctype.h	strftime	time.h
iscntrl	ctype.h	strlwr	string.h
isdigit	ctype.h	strupr	string.h
isgraph	ctype.h	strxfrm	string.h
islower	ctype.h	tolower	ctype.h
isprint	ctype.h	toupper	ctype.h
ispunct	ctype.h	vfprintf	stdio.h
isspace	ctype.h	vfscanf	stdio.h
isupper	ctype.h	vprintf	stdio.h
isxdigit	ctype.h	vscanf	stdio.h
localeconv	locale.h	vsprintf	stdio.h
printf	stdio.h	vsscanf	stdio.h

Manipulation Routines

See also

The following routines handle strings and blocks of memory: copying, comparing, converting, and searching.

Function	Header	Function	Header
mblen	stdlib.h	strerror	string.h
mbstowcs	stdlib.h	stricmp	string.h
mbtowc	stdlib.h	strlen	string.h
memccpy	mem.h, string.h	strlwr	string.h
memchr	mem.h, string.h	strncat	string.h
memcmp	mem.h, string.h	strncmpi	string.h
memcpy	mem.h, string.h	strncmp	string.h
memicmp	mem.h, string.h	strncpy	string.h
memmove	mem.h, string.h	strnicmp	string.h
memset	mem.h, string.h	strnset	string.h
movedata	mem.h, string.h	strpbrk	string.h
movmem	mem.h, string.h	strrchr	string.h
setmem	mem.h	strrev	string.h
stpcpy	string.h	strset	string.h
strcat	string.h	strspn	string.h
strchr	string.h	strstr	string.h
strcmpi	string.h	strtok	string.h
strcmp	string.h	strupr	string.h
strcoll	string.h	strxfrm	string.h
strcpy	string.h	wcstombs	stdlib.h
strcspn	string.h	wctomb	stdlib.h
strdup	string.h		

Math Routines

See also

The folowing routines perform mathematical calculations and conversions.

Function	Header	Function	Header
abs	complex.h, stdlib.h	labs	stdlib.h
acos	complex.h, math.h	ldexp	math.h
acosl	math.h	ldexpl	math.h
arg	complex.h	ldiv	math.h
asin	complex.h, math.h	log	complex.h, math.h
asinl	math.h	logl	math.h
atan	complex.h, math.h	log10	complex.h, math.h
atan2	complex.h, math.h	log10l	math.h
atan2l	math.h	_lrotl	stdlib.h
atanl	math.h	_lrotr	stdlib.h
atof	stdlib.h, math.h	ltoa	stdlib.h
atoi	stdlib.h	_matherr	math.h
atol	stdlib.h	_matherrl	math.h
_atold	math.h	modf	math.h
bcd (class)	bcd.h	modfl	math.h
cabs	math.h	norm	complex.h
cabsl	math.h	polar	complex.h
ceil	math.h	poly	math.h
ceill	math.h	polyl	math.h
_clear87	float.h	pow	complex.h, math.h
complex (class)	complex.h	pow10	math.h
conj	complex.h	pow10l	math.h
_control87	float.h	powl	math.h
cos	complex.h, math.h	rand	stdlib.h
cosh	complex.h, math.h	random	stdlib.h
coshl	math.h	randomize	stdlib.h
cosl	math.h	real	complex.h
div	math.h	_rotl	stdlib.h
ecvt	stdlib.h	_rotr	stdlib.h
exp	complex.h, math.h	sin	complex.h, math.h
expl	math.h	sinh	complex.h, math.h
fabs	math.h	sinhl	math.h
fabsl	math.h	sinl	complex.h, math.h
fcvt	stdlib.h	sqrt	complex.h, math.h
floor	math.h	sqrtl	math.h
floorl	math.h	srand	stdlib.h

fmod	math.h	_status87	float.h
fmodl	math.h	strtod	stdlib.h
_fpreset	float.h	strtol	stdlib.h
frexp	math.h	_strtold	stdlib.h
frexpl	math.h	strtoul	stdlib.h
gcvt	stdlib.h	tan	complex.h, math.h
hypot	math.h	tanh	complex.h, math.h
hypotl	math.h	tanhl	complex.h, math.h
imag	complex.h	tanl	math.h
itoa	stdlib.h	ultoa	stdlib.h

Memory Routines

See also

The following routines provide dynamic memory allocation in the small-data and large-data models.

Function	Header	Function	Header
alloca	malloc.h	heapcheckfree	alloc.h
_bios_memsize	bios.h	heapchecknode	alloc.h
calloc	alloc.h, stdlib.h	heapwalk	alloc.h
farcalloc	alloc.h	malloc	alloc.h, stdlib.h
farfree	alloc.h	realloc	alloc.h, stdlib.h
farmalloc	alloc.h	set_new_handler	new.h
free	alloc.h, stdlib.h	stackavail	malloc.h
heapcheck	alloc.h		

Miscellaneous Routines

See also

The following routines provide non-local goto capabilities and locale.

Function	Header
localeconv	locale.h
longjmp	setjmp.h
setjmp	setjmp.h
setlocale	locale.h

Obsolete Functions

See also

The old names of the following functions are available, but the compiler will generate a warning that you are using an obsolete name. Future versions of Borland C++ might not provide support for the old function names.

The following function names have been changed:

Old name	New name	Header file
_chmod	__rtl_chmod	io.h
_close	__rtl_close	io.h
_creat	__rtl_creat	io.h
_heapwalk	__rtl_heapwalk	malloc.h
_open	__rtl_open	io.h
_read	__rtl_read	io.h
_write	__rtl_write	io.h

Process Control Routines

See also

The following routines invoke and terminate new processes from within another routine.

Function	Header	Function	Header
abort	(process.h)	exit	(process.h)
_beginthread		_expand	(process.h)
_beginthreadNT	(process.h)	getpid	(process.h)
_c_exit	(process.h)	_pclose	(stdio.h)
_cexit	(process.h)	_popen	(stdio.h)
cwait	(process.h)	raise	(signal.h)
_endthread	(process.h)	signal	(signal.h)
execle	(process.h)	spawnle	(process.h)
execl	(process.h)	spawnlpe	(process.h)
execlpe	(process.h)	spawnlp	(process.h)
execlp	(process.h)	spawnl	(process.h)
execve	(process.h)	spawnve	(process.h)
execv	(process.h)	spawnvpe	(process.h)
execvpe	(process.h)	spawnvp	(process.h)
execvp	(process.h)	spawnv	(process.h)
_exit	(process.h)	wait	(process.h)

Time and Date Routines

See also

The following following functions are time conversion and time manipulation routines.

Function	Header	Function	Header
asctime	time.h	gmtime	time.h
_bios_timeofday	bios.h	localtime	time.h
ctime	time.h	mktime	time.h
difftime	time.h	stime	time.h
_dos_getdate	dos.h	_strdate	time.h
_dos_gettime	dos.h	strftime	time.h
_dos_setdate	dos.h	_strtime	time.h
_dos_settime	dos.h	TDate (class)	date.h
dostounix	dos.h	time	time.h
ftime	sys\timeb.h	TTime (class)	date.h
getdate	dos.h	tzset	time.h
gettime	dos.h	unixtodos	dos.h

Variable Argument List Routines

See also

The following routines are for use when accessing variable argument lists (such as with printf, vscanf, and so on).

Function	Header
va_start	stdarg.h
va_arg	stdarg.h
va_end	stdarg.h

Borland C++ Library Routines, by Name

See also Overview

If you do not know the name of a particular function, but you know what type of action it performs, see:

Borland C++ Library Routines, by Category

Otherwise, if you know which function you want Help on, choose one of the following topics:

A

abort
abs
access
acos
acosl
alloca
arg
asctime
asin
asinl
assert
atan
atan2
atan2l
atanl
atexit
atof
atoi
atol
_atold

B

bcd (class)
bdos
bdosptr
_beginthread
_beginthreadNT
_bios_equiplist
_bios_memsize
_bios_timeofday
biosequip
biosmemory
biostime
bsearch

C

_c_exit
cabs
cabsl
calloc

ceil
ceill
_cexit
cgets
_chain_intr
chdir
_chdrive
chmod
chsize
_clear87
clearerr
clock
close
closedir
clreol
clrscr
complex (class)
conj
_control87
cos
cosh
coshl
cosl
country
creat
creatnew
creattemp
_crotl
_crotr
cscanf
ctime
ctrlbrk
cwait

D

delline
difftime
disable
_disable
div
_dos_close
_dos_commit
_dos_creat
_dos_creatnew
_dos_findfirst
_dos_findnext
_dos_getdate
_dos_getdiskfree
_dos_getdrive
_dos_getfileattr

_dos_getftime
_dos_gettime
_dos_getvect
_dos_open
_dos_read
_dos_setdate
_dos_setdrive
_dos_setfileattr
_dos_setftime
_dos_settime
_dos_setvect
_dos_write
dosexterr
dostounix
dup

E

ecvt
__emit__
enable
_enable
_endthread
eof
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe
exit
_exit
exp
_expand
expl

F

fabs
fabsl
farcalloc
farfree
farmalloc
farrealloc
fclose
fcloseall
fcvt
fdopen
feof
ferror

fflush
fgetc
fgetchar
fgetpos
fgets
filelength
fileno
findfirst
findnext
floor
floorl
flushall
_fmemccpy
_fmemchr
_fmemcmp
_fmemcpy
_fmemicmp
_fmemset
fmod
fmodl
fnmerge
fnsplit
fopen
FP_OFF
FP_SEG
_fpreset
fprintf
fputc
fputchar
fputs
fread
free
freopen
frexp
frexpl
fscanf
fseek
fsetpos
_fsopen
fstat
_fstrcat
_fstrchr
_fstrcspn
_fstrdup
_fstricmp
_fstrlen
_fstrlwr
_fstrncat
_fstrncmp

_fstrncpy
_fstrnicmp
_fstrnset
_fstrpbrk
_fstrrchr
_fstrrev
_fstrset
_fstrspn
_fstrstr
_fstrtok
_fstrupr
ftell
ftime
_fullpath
fwrite

G

gcvt
geninterrupt
_get_osfhandle
getc
getcbrk
getch
getchar
getche
getcurdir
getcwd
getdate
_getdcwd
getdfree
getdisk
getdta
getenv
getfat
getfatd
getftime
getpass
getpid
getpsp
gets
gettext
gettextinfo
gettime
getvect
getverify
getw
gmtime
gotoxy

H

heapcheck
heapcheckfree
heapchecknode
_heapchk
heapfillfree
_heapmin
_heapset
heapwalk
highvideo
hypot
hypotl

I

imag
_InitEasyWin
inp
inport
inportb
inpw
insline
int86x
intdos
intdosx
intr
ioctl
isalnum
isalpha
isascii
isatty
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
itoa

K

kbhit

L

labs
ldexp
ldexpl
ldiv
lfind
localeconv
localtime

lock
locking
log
log10
log10l
logl
longjmp
lowvideo
_lrotl
_lrotr
lsearch
lseek
ltoa

M

_makepath
malloc
_matherr
_matherrl
max
mblen
mbstowcs
mbtowc
memccpy
memchr
memcmp
memcpy
memicmp
memmove
memset
min
MK_FP
mkdir
mktemp
mktime
modf
modfl
movedata
movetext
movmem
_msize

N

norm
normvideo

O

offsetof
open
_open_osfhandle
opendir

outp
outport
outportb
outpw

P

parsfnm
_pclose
peek
peekb
perror
_pipe
poke
pokeb
polar
poly
polyl
_popen
pow
pow10
pow10l
powl
printf
putc
putch
putchar
putenv
puts
puttext
putw

Q

qsort

R

raise
rand
random
randomize
read
readdir
real
realloc
remove
rename
rewind
rewinddir
rmdir
rmtmp
_rotl
_rotr

_rtl_chmod
_rtl_close
_rtl_creat
_rtl_heapwalk
_rtl_open
_rtl_read
_rtl_write

S

scanf
_searchenv
searchpath
_searchstr
segread
set_new_handler
setbuf
setcbrk
_setcursortype
setdate
setdisk
setdta
setftime
setjmp
setlocale
setmem
setmode
settime
setvbuf
setvect
setverify
signal
sin
sinh
sinhl
sinl
sleep
sopen
spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe
_splitpath
sprintf
sqrt
sqrtl
srand

sscanf
stackavail
stat
_status87
stime
stpcpy
strcat
strchr
strcmp
strcmpi
strcoll
strcpy
strcspn
_strdate
strdup
strerror
_strerror
strftime
stricmp
strlen
strlwr
strncat
strncmp
strncmpi
strncpy
strnicmp
strnset
strpbrk
strrchr
strrev
strset
strspn
strstr
_strtime
strtod
strtok
strtol
_strtold
strtoul
strupr
strxfrm
swab
system

T

tan
tanh
tanhl
tanl
tell

tempnam
textattr
textbackground
textcolor
textmode
time
tmpfile
tmpnam
toascii
tolower
_tolower
toupper
_toupper
tzset

U

ultoa
umask
ungetc
ungetch
unixtodos
unlink
unlock
utime

V

va_arg
va_end
va_start
vfprintf
vfscanf
vprintf
vscanf
vsprintf
vsscanf

W

wait
wcstombs
wctomb
wherex
wherey
window
write

abort

See also Example Portability

Syntax

```
#include <stdlib.h>
void abort(void);
```

Description

Abnormally terminates a program.

abort causes an abnormal program termination by calling *raise*(SIGABRT). If there is no signal handler for SIGABRT, then *abort* writes a termination message (Abnormal program termination) on stderr, then aborts the program by a call to *_exit* with exit code 3.

Return Value

abort returns the exit code 3 to the parent process or to the operating system command processor.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

abs

See also Example Portability

Syntax

```
#include <stdlib.h>
int abs(int x);
```

Description

Returns the absolute value of an integer.

abs returns the absolute value of the integer argument *x*. If *abs* is called when stdlib.h has been included, it's treated as a macro that expands to inline code.

If you want to use the *abs* function instead of the macro, include

```
#undef abs
```

in your program, after the #include <stdlib.h>.

This function can be used with bcd and complex types.

Return Value

The *abs* function returns an integer in the range of 0 to INT_MAX, with the exception that an argument with the value INT_MIN is returned as INT_MIN. The values for INT_MAX and INT_MIN are defined in header file limit.h.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

access, _waccess

See also Example Portability

Syntax

```
#include <io.h>
int access(const char *filename, int amode);
int _waccess(const wchar_t *filename, int amode);
```

Description

Determines accessibility of a file.

access checks the file named by *filename* to determine if it exists, and whether it can be read, written to, or executed.

The list of *amode* values is as follows:

- 06 Check for read and write permission
- 04 Check for read permission
- 02 Check for write permission
- 01 Execute (ignored)
- 00 Check for existence of file

Under DOS, OS/2, and Windows (16- and 32-bit) all existing files have read access (*amode* equals 04), so 00 and 04 give the same result. Similarly, *amode* values of 06 and 02 are equivalent because under DOS write access implies read access.

If *filename* refers to a directory, *access* simply determines whether the directory exists.

Return Value

If the requested access is allowed, *access* returns 0; otherwise, it returns a value of -1, and the global variable errno is set to one of the following values:

ENOENT	Path or file name not found
EACCES	Permission denied

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

acos, acosl

See also Example Portability

Syntax

```
#include <math.h>
double acos(double x);
long double acosl(long double x);
```

Description

Calculates the arc cosine.

acos returns the arc cosine of the input value.

acosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Arguments to *acos* and *acosl* must be in the range -1 to 1, or else *acos* and *acosl* return NAN and set the global variable errno to

EDOM Domain error

This function can be used with bcd and complex types.

Return Value

acos and *acosl* of an argument between -1 and +1 return a value in the range 0 to *pi*. Error handling for these routines can be modified through the functions _matherr_matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
acos	+	+	+	+	+	+	+
acosl	+		+	+			+

alloca

See also Example Portability

Syntax

```
#include <malloc.h>
void *alloca(size_t size);
```

Description

Allocates temporary stack space.

alloca allocates *size* bytes on the stack; the allocated space is automatically freed up when the calling function exits.

Because *alloca* modifies the stack pointer, do not place calls to *alloca* in an expression that is an argument to a function.

The *alloca* function should not be used in the **try**-block of a C++ program. If an exception is thrown, any values placed on the stack by *alloca* will be corrupted.

If the calling function does not contain any references to local variables in the stack, the stack will not be restored correctly when the function exits, resulting in a program crash. To ensure that the stack is restored correctly, use the following code in the calling function:

```
char *p;
char dummy[5];

dummy[0] = 0;

  .
  .
  .
p = alloca(nbytes);
```

Return Value

If enough stack space is available, *alloca* returns a pointer to the allocated stack area. Otherwise, it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

asctime, _wasctime

See also Example Portability

Syntax

```
#include <time.h>
char *asctime(const struct tm *tblock);
wchar_t *_wasctime(const struct tm *tblock);
```

Description

asctime converts date and time to ASCII.

_wasctime converts date and time to a **wchar_t** string.

asctime converts a time stored as a structure in **tblock* to a 26-character string of the same form as the *ctime* string:

```
Sun Sep 16 01:03:52 1973\n\0
```

Return Value

asctime returns a pointer to the character string containing the date and time. This string is a static variable that is overwritten with each call to *asctime*.

Valid values for **struct** *tm* are as follows:

`tm.day`	0 - 6	0 = Sunday
`tm.month`	0 - 11	0 = January

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

asin, asinl

See also Example Portability

Syntax

```
#include <math.h>
double asin(double x);
long double asinl(long double x);
```

Description

Calculates the arc sine.

asin of a real argument returns the arc sine of the input value.

asinl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Real arguments to *asin* and *asinl* must be in the range -1 to 1, or else *asin* and *asinl* return NAN and set the global variable errno to

EDOM Domain error

This function can be used with *bcd* and *complex* types.

Return Value

asin and *asinl* of a real argument return a value in the range -*pi*/2 to *pi*/2. Error handling for these functions may be modified through the functions *_matherr* and *_matherrl*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
asin	+	+	+	+	+	+	+
asinl	+		+	+			+

assert

See also Example Portability

Syntax

```
#include <assert.h>
void assert(int test);
```

Description

Tests a condition and possibly aborts.

assert is a macro that expands to an **if** statement; if *test* evaluates to zero, *assert* aborts the program (by calling abort) and asserts the following a message on stderr:

```
Assertion failed: test, file filename, line linenum
```

The *filename* and *linenum* listed in the message are the source file name and line number where the *assert* macro appears.

If you place the #define NDEBUG directive ("no debugging") in the source code before the #include <assert.h> directive, the effect is to comment out the assert statement.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

atan, atanl

See also Example Portability

Syntax

```
#include <math.h>
double atan(double x);
long double atanl(long double x);
```

Description

Calculates the arc tangent.

atan calculates the arc tangent of the input value.

atanl is the **long double** version; it takes a **long double** argument and returns a **long double** result. This function can be used with *bcd* and *complex* types.

Return Value

atan and *atanl* of a real argument return a value in the range -*pi*/2 to *pi*/2. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl.*

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atan	+	+	+	+	+	+	+
atanl	+		+	+			+

atan2, atan2l

See also Example Portability

Syntax

```
#include <math.h>
double atan2(double y, double x);
long double atan2l(long double y, long double x);
```

Description

Calculates the arc tangent of *y/x*.

atan2 returns the arc tangent of *y/x*; it produces correct results even when the resulting angle is near *pi*/2 or -*pi*/2 (*x* near 0). If both *x* and *y* are set to 0, the function sets the global variable errno to EDOM, indicating a domain error.

atan2l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

atan2 and *atan2l* return a value in the range -*pi* to *pi*. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atan2	+	+	+	+	+	+	+
atan2l	+		+	+			+

atexit

See also Example Portability

Syntax

```
#include <stdlib.h>
int atexit(void (_USERENTRY * func)(void));
```

Description

Registers termination function.

atexit registers the function pointed to by *func* as an exit function. Upon normal termination of the program, *exit* calls *func* just before returning to the operating system. fcmp must be used with the _USERENTRY calling convention.

Each call to *atexit* registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed).

Return Value

atexit returns 0 on success and nonzero on failure (no space left to register the function).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

atof, _atold, _wtof, _wtold

See also Example Portability

Syntax

```
#include <math.h>
double atof(const char *s);
double _wtof(const wchar_t *s);
long double _atold(const char *s);
long double _wtold(const wchar_t *s);
```

Description

Converts a string to a floating-point number.

atof converts a string pointed to by *s* to **double**; this function recognizes the character representation of a floating-point number, made up of the following:

- An optional string of tabs and spaces
- An optional sign
- A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
- An optional *e* or *E* followed by an optional signed integer

The characters must match this generic format:

[whitespace] [sign] [ddd] [.] [ddd] [e|E[sign]ddd]

atof also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for Not-a-Number.

In this function, the first unrecognized character ends the conversion.

_atold is the l**ong double** version; it converts the string pointed to by *s* to a **long double.**

The functions strtod and _strtold are similar to *atof* and *_atold*; they provide better error detection, and hence are preferred in some applications.

Return Value

atof and *_atold* return the converted value of the input string.

If there is an overflow, *atof* (or *_atold*) returns plus or minus HUGE_VAL (or _LHUGE_VAL), errno is set to ERANGE (Result out of range), and _matherr (or _matherrl) is not called.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atof	+	+	+	+	+	+	+
_atold	+		+	+			+

atoi, _atoi64, _wtoi, _wtoi64

See also Example Portability

Syntax

```
#include <stdlib.h>
int atoi(const char *s);
__int64 _atoi64(const char *s);
int _wtoi(const wchar_t *s);
__int64 _wtoi64(const wchar_t *s);
```

Description

Converts a string to an integer.

atoi converts a string pointed to by *s* to **int**; *atoi* recognizes (in the following order)

- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

[ws] [sn] [ddd]

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in *atoi* (results are undefined).

Return Value

atoi returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**int**), *atoi* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

atol, _wtol

See also Example Portability

Syntax

```
#include <stdlib.h>
long atol(const char *s);
long _wtol(const wchar_t *s);
```

Description

Converts a string to a long.

atol converts the string pointed to by *s* to **long**. *atol* recognizes (in the following order)

- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

```
[ws] [sn] [ddd]
```

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in *atol* (results are undefined).

Return Value

atol returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (b), *atol* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

bdos

See also Example Portability

Syntax

```
#include <dos.h>
int bdos(int dosfun, unsigned dosdx, unsigned dosal);
```

Description

Accesses DOS system calls.

bdos provides direct access to many of the DOS system calls. See your DOS reference manuals for details on each system call.

For system calls that require an integer argument, use *bdos*; if they require a pointer argument, use *bdosptr*. In the large data models (compact, large, and huge), it is important to use bdosptr instead of *bdos* for system calls that require a pointer as the call argument.

- *dosfun* is defined in your DOS reference manuals.
- *dosdx* is the value of register DX.
- *dosal* is the value of register AL.

Return Value

The return value of *bdos* is the value of AX set by the system call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

bdosptr

See also Example Portability

Syntax

```
#include <dos.h>
int bdosptr(int dosfun, void *argument, unsigned dosal);
```

Description

Accesses DOS system calls.

bdosptr provides direct access to many of the DOS system calls. See your DOS reference manuals for details of each system call.

For system calls that require an integer argument, use *bdos*; if calls require a pointer argument, use *bdosptr*. In the large data models (compact, large, and huge), it is important to use *bdosptr* for system calls that require a pointer as the call argument. In the small data models, the *argument* parameter to *bdosptr* specifies DX; in the large data models, it gives the DS:DX values to be used by the system call.

dosfun is defined in your DOS reference manuals. *dosal* is the value of register AL.

Return Value

The return value of *bdosptr* is the value of AX on success or -1 on failure. On failure, the global variables errno and _doserrno are set.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_beginthread

See also Example Portability

Syntax

```
#include <process.h>
unsigned long _beginthread(_USERENTRY (*start_address)(void *), unsigned
  stack_size, void *arglist)
```

Description

Starts execution of a new thread.

Note: The *start_address* must be declared to be *_USERENTRY*.

The *_beginthread* function creates and starts a new thread. The thread starts execution at *start_address*.

The size of its stack in bytes is *stack_size*; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread is passed *arglist* as its only parameter; it can be NULL, but must be present. The thread terminates by simply returning, or by calling _endthread.

Either this function or _beginthreadNT must be used instead of the operating system thread-creation API function because *_beginthread* and *_beginthreadNT* perform initialization required for correct operation of the run-time library functions.

This function is available only in the multithread libraries.

Return Value

_beginthread returns the handle of the new thread.

On error, the function returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid request

Also see the Win32 description of *GetLastError*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

_beginthreadNT

See also Example Portability

Syntax

```
#include <process.h>
unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *),
  unsigned stack_size, void *arglist, void *security_attrib, unsigned long
  create_flags, unsigned long *thread_id);
```

Description

Starts execution of a new thread under Windows NT.

Note: The start_address must be declared to be *_USERENTRY*.

All multithread Windows NT programs must use *_beginthreadNT* or the _beginthreadfunction instead of the operating system thread-creation API function because these functions perform initialization required for correct operation of the run-time library functions. The *_beginthreadNT* function provides support for the operating system security. These functions are available only in the multithread libraries.

The *_beginthreadNT* function creates and starts a new thread. The thread starts execution at *start_address*.

The size of its stack in bytes is *stack_size*; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread *arglist* can be NULL, but must be present. The thread terminates by simply returning, or by calling *_endthread*.

The *_beginthreadNT* function uses the *security_attr* pointer to access the SECURITY_ATTRIBUTES structure. The structure contains the security attributes for the thread. If *security_attr* is NULL, the thread is created with default security attributes. The thread handle is not inherited if *security_attr* is NULL.

_beginthreadNT reads the *create_flags* variable for flags that provide additional information about the thread creation. This variable can be zero, specifying that the thread will run immediately upon creation. The variable can also be CREATE_SUSPENDED; in which case, the thread will not run until the *ResumeThread* function is called. *ResumeThread* is provided by the Win32 API.

_beginthreadNT initializes the *thread_id* variable with the thread identifier.

Return Value

On success, *_beginthreadNT* returns the handle of the new thread.

On error, it returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid request

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			

biosequip

Example Portability

Syntax

```
#include <bios.h>
int biosequip(void);
```

Description

Checks equipment.

biosequip uses BIOS interrupt 0x11 to return an integer describing the equipment connected to the system.

Return Value

The return value is interpreted as a collection of bit-sized fields. The IBM PC values follow:

Bits 14-15	Number of parallel printers installed 00 = 0 printers 01 = 1 printer 10 = 2 printers 11 = 3 printers
Bit 13	Serial printer attached
Bit 12	Game I/O attached
Bits 9-11	Number of COM ports (DOS only sees two ports but can be pushed to see four; the IBM PS/2 can see up to eight.) 000 = 0 ports 001 = 1 port 010 = 2 ports 011 = 3 ports 100 = 4 ports 101 = 5 ports 110 = 6 ports 111 = 7 ports
Bit 8	Direct memory access (DMA) 0 = Machine has DMA 1 = Machine does not have DMA; for example, PC Jr.
Bits 6-7	Number of disk drives 00 = 1 drive 01 = 2 drives 10 = 3 drives 11 = 4 drives, only if bit 0 is 1
Bits 4-5	Initial video mode 00 = Unused 01 = 40x25 BW with color card 10 = 80x25 BW with color card 11 = 80x25 BW with mono card
Bits 2-3	Motherboard RAM size 00 = 16K 01 = 32K 10 = 48K 11 = 64K

Bit 1	Floating-point coprocessor
Bit 0	Boot from disk

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_bios_equiplist

Example Portability

Syntax

```
#include <bios.h>
unsigned _bios_equiplist(void);
```

Description

Checks equipment.

_bios_equiplist uses BIOS interrupt 0x11 to return an integer describing the equipment connected to the system.

Return Value

The return value is interpreted as a collection of bit-sized fields. The IBM PC values follow:

Bits 14-15	Number of parallel printers installed 00 = 0 printers 01 = 1 printer 10 = 2 printers 11 = 3 printers
Bit 13	Serial printer attached
Bit 12	Game I/O attached
Bits 9-11	Number of COM ports (DOS only sees two ports but can be pushed to see four; the IBM PS/2 can see up to eight.) 000 = 0 ports 001 = 1 port 010 = 2 ports 011 = 3 ports 100 = 4 ports 101 = 5 ports 110 = 6 ports 111 = 7 ports
Bit 8	Direct memory access (DMA) 0 = Machine has DMA 1 = Machine does not have DMA; for example, PC Jr.
Bits 6-7	Number of disk drives 00 = 1 drive 01 = 2 drives 10 = 3 drives 11 = 4 drives, only if bit 0 is 1
Bits 4-5	Initial video mode 00 = Unused 01 = 40x25 BW with color card 10 = 80x25 BW with color card 11 = 80x25 BW with mono card
Bits 2-3	Motherboard RAM size 00 = 16K 01 = 32K 10 = 48K 11 = 64K

Bit 1	Floating-point coprocessor
Bit 0	Boot from disk

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

biosmemory

Example Portability

Syntax

```
#include <bios.h>
int biosmemory(void);
```

Description

Returns memory size.

biosmemory returns the size of RAM memory using BIOS interrupt 0x12. This does not include display adapter memory, extended memory, or expanded memory.

Return Value

biosmemory returns the size of RAM memory in 1K blocks.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_bios_memsize

Example Portability

Syntax

```
#include <bios.h>
unsigned _bios_memsize(void);
```

Description

Returns memory size.

_bios_memsize returns the size of RAM memory using BIOS interrupt 0x12. This does not include display adapter memory, extended memory, or expanded memory.

Return Value

_bios_memsize returns the size of RAM memory in 1K blocks.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

biostime

Example Portability

Syntax

```
#include <bios.h>
long biostime(int cmd, long newtime);
```

Description

Reads or sets the BIOS timer.

biostime either reads or sets the BIOS timer. This is a timer counting ticks since midnight at a rate of ticks per second as defined by _BIOS_CLOCKS_PER_SEC. *biostime* uses BIOS interrupt 0x1A.

If *cmd* equals 0, *biostime* returns the current value of the timer. If *cmd* equals 1, the timer is set to the **long** value in *newtime*. For example:

```
totalsecs = biostime(int cmd, long newtime) / _BIOS_CLK_TCK;
```

The _BIOS_CLOCKS_PER_SEC and _BIOS_CLK_TCK constants are defined in bios.h.

Return Value

When *biostime* reads the BIOS timer (*cmd* = 0), it returns the timer's current value.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_bios_timeofday

Example Portability

Syntax

```
#include <bios.h>
unsigned _bios_timeofday(int cmd, long *timep);
```

Description

Reads or sets the BIOS timer.

_bios_timeofday either reads or sets the BIOS timer. This is a timer counting ticks since midnight at a rate of roughly 18.2 ticks per second. *_bios_timeofday* uses BIOS interrupt 0x1A.

The *cmd* parameter can be either of the following values:

_TIME_GETCLOCK	The function stores the current BIOS timer value into the location pointed to by *timep*. If the timer has not been read or written since midnight, the function returns 1. Otherwise, the function returns 0.
_TIME_SETCLOCK	The function sets the BIOS timer to the long value pointed to by *timep*. The function does not return a value.

Return Value

The *_bios_timeofday* returns the value in AX that was set by the BIOS timer call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

bsearch

See also Example Portability

Syntax

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nelem, size_t width,
  int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Binary search of an array.

bsearch searches a table (array) of *nelem* elements in memory, and returns the address of the first entry in the table that matches the search key. The array must be in order. If no match is found, *bsearch* returns 0.

Note: Because this is a binary search, the first matching entry is not necessarily the first entry in the table.

The type *size_t* is defined in stddef.h header file.

- *nelem* gives the number of elements in the table.
- *width* specifies the number of bytes in each table entry.

The comparison routine *fcmp* must be used with the _USERENTRY calling convention.

fcmp is called with two arguments: *elem1* and *elem2*. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (**elem1* and **elem2*), and returns an integer based on the results of the comparison.

For *bsearch*, the *fcmp* return value is

- < 0 if **elem1* < **elem2*
- == 0 if **elem1* == **elem2*
- > 0 if **elem1* > **elem2*

Return Value

bsearch returns the address of the first entry in the table that matches the search key. If no match is found, *bsearch* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

cabs, cabsl

See also Example Portability

Syntax

```
#include <math.h>
double cabs(struct complex z);
long double cabsl(struct _complexl z);
```

Description

cabs calculates the absolute value of a complex number. cabs is a macro that calculates the absolute value of *z*, a complex number. *z* is a structure with type *complex*; the structure is defined in math.h as

```
struct complex {
   double x, y;
   };
```

where *x* is the real part, and *y* is the imaginary part.

Calling *cabs* is equivalent to calling *sqrt* with the real and imaginary components of *z*, as shown here:

```
sqrt(z.x * z.x + z.y * z.y)
```

cabsl is the **long double** version; it takes a structure with *type _complexl* as an argument, and returns a **long double** result. The structure is defined in math.h as

```
struct _complexl {
   long double x, y;
};
```

Note: If you are using C++, you may also use the complex class defined in complex.h, and use the function *abs* to get the absolute value of a *complex* number.

Return Value

cabs (or *cabsl*) returns the absolute value of *z*, a double. On overflow, *cabs* (or *cabsl*) returns HUGE_VAL (or _LHUGE_VAL) and sets the global variable errno to

ERANGE	Result out of range

Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cabs	+	+	+	+			+
cabsl	+		+	+			+

calloc

See also Example Portability

Syntax

```
#include <stdlib.h>
void *calloc(size_t nitems, size_t size);
```

Description

Allocates main memory.

calloc provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

All the space between the end of the data segment and the top of the program stack is available for use in the tiny (DOS only), small and medium data models,except for a small margin immediately before the top of the stack. This margin allows room for the application to grow on the stack, and provides a small amount of room needed by the operating system.

In the large data models (compact, large, and huge), all space beyond the program stack to the end of physical memory is available for the heap.

Note: Memory models are available only for 16-bit applications.

calloc allocates a block of size *nitems* * *size*. The block is cleared to 0. If you want to allocate a block larger than 64K, you must use *farcalloc*.

Return Value

calloc returns a pointer to the newly allocated block. If not enough space exists for the new block or if *nitems* or *size* is 0, *calloc* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ceil, ceill

See also Example Portability

Syntax

```
#include <math.h>
double ceil(double x);
long double ceill(long double x);
```

Description

Rounds up.

ceil finds the smallest integer not less than *x*.

ceill is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

These functions return the integer found as a **double** *(ceil)* or a **long double** *(ceill)*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
ceil	+	+	+	+	+	+	+
ceill	+		+	+			+

_c_exit

See also Example Portability

Syntax

```
#include <process.h>
void _c_exit(void);
```

Description

Performs _exit cleanup without terminating the program.

_c_exit performs the same cleanup as _exit, except that it does not terminate the calling process.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_cexit

See also Example Portability

Syntax

```
#include <process.h>
void _cexit(void);
```

Description

Performs exit cleanup without terminating the program.

_cexit performs the same cleanup as exit, closing all files but without terminating the calling process. The *_cexit* function calls any registered "exit functions" (posted with *atexit*). Before *_cexit* returns, it flushes all input/output buffers and closes all streams which were open.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

cgets

See also Example Portability

Syntax

```
#include <conio.h>
char *cgets(char *str);
```

Description

Reads a string from the console.

cgets reads a string of characters from the console, storing the string (and the string length) in the location pointed to by *str*.

cgets reads characters until it encounters a carriage-return/linefeed (CR/LF) combination, or until the maximum allowable number of characters have been read. If *cgets* reads a CR/LF combination, it replaces the combination with a \0 (null terminator) before storing the string.

Before *cgets* is called, set str[0] to the maximum length of the string to be read. On return, *str*[1] is set to the number of characters actually read. The characters read start at *str*[2] and end with a null terminator. Thus, *str* must be at least *str*[0] plus 2 bytes long.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

On success, *cgets* returns a pointer to *str*[2].

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_chain_intr

See also Example Portability

Syntax

```
#include <dos.h>
void _chain_intr(void (interrupt far *newhandler)());
```

Description

Chains to another interrupt handler.

_chain_intr passes control from the currently executing interrupt handler to the new interrupt handler whose address is *newhandler*. The current register set is *not* passed to the new handler. Instead, the new handler receives the registers that were stacked (and possibly modified in the stack) by the old handler. The new handler can simply return, as if it were the original handler. The old handler is not entered again.

_chain_intr can be called only by C interrupt functions. It is useful when writing a TSR that needs to insert itself in a chain of interrupt handlers (such as the keyboard interrupt).

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

chdir, _wchdir

See also Example Portability

Syntax

```
#include <dir.h>
int chdir(const char *path);
int _wchdir(const wchar_t *path);
```

Description

Changes current directory.

chdir causes the directory specified by *path* to become the current working directory; *path* must specify an existing directory.

A drive can also be specified in the path argument, such as

```
chdir("a:\\BC")
```

but this method changes only the current directory on that drive; it does not change the active drive.

Under Windows, only the current process is affected.

Under DOS, the function changes the current directory of the parent process.

Return Value

Upon successful completion, the functions return a value of 0. Otherwise, they return a value of -1, and the global variable errno is set to

ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_chdrive

See also Example Portability

Syntax

```
#include <direct.h>
int _chdrive(int drive);
```

Description

Sets current disk drive.

_chdrive sets the current drive to the one associated with *drive:* 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

Return Value

_chdrive returns 0 if the current drive was changed successfully; otherwise, it returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

chmod, _wchmod

See also Example Portability

Syntax

```
#include <io.h>
int chmod(const char *path, int amode);
int _wchmod(const wchar_t *path, int amode);
```

Description

Changes file access mode.

chmod sets the file-access permissions of the file given by *path* according to the mask given by *amode. path* points to a string.

amode can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\ stat.h).

Value of amode	Access permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD | S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successfully changing the file access mode, *chmod* returns 0. Otherwise, *chmod* returns a value of -1.

In the event of an error, the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

chsize

See also Example Portability

Syntax

```
#include <io.h>
int chsize(int handle, long size);
```

Description

Changes the file size.

chsize changes the size of the file associated with *handle*. It can truncate or extend the file, depending on the value of *size* compared to the file's original size.

The mode in which you open the file must allow writing.

If *chsize* extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

Return Value

On success, *chsize* returns 0. On failure, it returns -1 and the global variable errno is set to one of the following values:

EACCESS	Permission denied
EBADF	Bad file number
ENOSPC	No space left on device

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_clear87

See also Example Portability

Syntax

```
#include <float.h>
unsigned int _clear87 (void);
```

Description

Clears floating-point status word.

_clear87 clears the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

Return Value

The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

clearerr

See also Example Portability

Syntax

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

Resets error indication.

clearerr resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to *clearerr* or *rewind*. The end-of-file indicator is reset with each input operation.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

clock

See also Example Portability

Syntax

```
#include <time.h>
clock_t clock(void);
```

Description

Determines processor time.

clock can be used to determine the time interval between two events. To determine the time in seconds, the value returned by *clock* should be divided by the value of the macro *CLK_TCK.*

Return Value

On success, *clock* returns the processor time elapsed since the beginning of the program invocation.

On error (if the processor time is not available or its value cannot be represented), *clock* returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

close

See also Example Portability

Syntax

```
#include <io.h>
int close(int handle);
```

Description

Closes a file.

The *close* function closes the file associated with *handle*, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open.

It does not write a *Ctrl-Z* character at the end of the file. If you want to terminate the file with a *Ctrl-Z*, you must explicitly output one.

Return Value

Upon successful completion, *close* returns 0.

On error (if it fails because *handle* is not the handle of a valid, open file), *close* returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

closedir, wclosedir

See also Example Portability

Syntax

```
#include <dirent.h>
int closedir(DIR *dirp);
int wclosedir(wDIR *dirp);
```

Description

Closes a directory stream.

On UNIX platforms, *closedir* is available on POSIX-compliant systems.

The *closedir* function closes the directory stream *dirp*, which must have been opened by a previous call to *opendir*. After the stream is closed, *dirp* no longer points to a valid directory stream.

wclosedir is the Unicode version of *closedir*.

Return Value

If *closedir* is successful, it returns 0. Otherwise, *closedir* returns -1 and sets the global variable errno to

EBADF	The *dirp* argument does not point to a valid open directory stream

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

clreol

See also Example Portability

Syntax

```
#include <conio.h.>
void clreol(void);
```

Description

Clears to end of line in text window.

clreol clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor.

Note: This function should not be used in Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

clrscr

See also Example Portability

Syntax

```
#include <conio.h>
void clrscr(void);
```

Description

Clears the text-mode window.

clrscr clears the current text window and places the cursor in the upper left corner (at position 1,1).

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_control87

See also Portability

Syntax

```
#include <float.h>
unsigned int _control87(unsigned int newcw, unsigned int mask);
```

Description

Manipulates the floating-point control word.

_control87 retrieves or changes the floating-point control word.

The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package; namely, the precision, infinity, and rounding modes. Changing these modes lets you mask or unmask floating-point exceptions.

_control87 matches the bits in *mask* to the bits in *newcw*. If a *mask* bit equals 1, the corresponding bit in *newcw* contains the new value for the same bit in the floating-point control word, and *_control87* sets that bit in the control word to the new value.

Here is a simple illustration:

Original control word:	0100	0011	0110	0011
mask:	1000	0001	0100	1111
newcw:	1110	1001	0000	0101
Changing bits:	1*xxx*	*xxx*1	*x*0*xx*	0101

If *mask* equals 0, *_control87* returns the floating-point control word without altering it.

Return Value

The bits in the value returned reflect the new floating-point control word. For a complete definition of the bits returned by *_control87*, see the header file float.h.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

cos, cosl

See also Example Portability

Syntax

```
#include <math.h>
double cos(double x);
long double cosl(long double x);
```

Description

Calculates the cosine of a value.

cos computes the cosine of the input value. The angle is specified in radians.

cosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with bcd and complex types.

Return Value

cos of a real argument returns a value in the range -1 to 1. Error handling for these functions can be modified through *_matherr* (or *_matherrl*).

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cos	+	+	+	+	+	+	+
cosl	+		+	+			+

cosh, coshl

See also Example Portability

Syntax

```
#include <math.h>
double cosh(double x);
long double coshl(long double x);
```

Description

Calculates the hyperbolic cosine of a value.

cosh computes the hyperbolic cosine, $(e^x + e^{-x})/2$. *coshl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

Return Value

cosh returns the hyperbolic cosine of the argument.

When the correct value would create an overflow, these functions return the value HUGE_VAL *(cosh)* or _LHUGE_VAL *(coshl)* with the appropriate sign, and the global variable errno is set to ERANGE. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl.*

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cosh	+	+	+	+	+	+	+
coshl	+		+	+			+

country

Example Portability

Syntax

```
#include <dos.h>
struct COUNTRY *country(int xcode, struct country *cp);
```

Description

Returns country-dependent information.

country specifies how certain country-dependent data (such as dates, times, and currency) will be formatted. The values set by this function depend on the operating system version being used.

If *cp* has a value of -1, the current country is set to the value of *xcode*, which must be nonzero. The *COUNTRY* structure pointed to by *cp* is filled with the country-dependent information of the current country (if *xcode* is set to zero), or the country given by *xcode*.

The structure *COUNTRY* is defined as follows:

```
struct COUNTRY{
   short co_date;             /* date format */
   char co_curr[5];           /* currency symbol */
   char co_thsep[2];          /* thousands separator */
   char co_desep[2];          /* decimal separator */
   char co_dtsep[2];          /* date separator */
   char co_tmsep[2];          /* time separator */
   char co_currstyle;         /* currency style */
   char co_digits;            /* significant digits in currency */
   char co_time;              /* time format */
   long co_case;              /* case map */
   char co_dasep[2];          /* data separator */
   char co_fill[10];          /* filler */
};
```

The date format in *co_date* is

0 for the U.S. style of month, day, year.

1 for the European style of day, month, year.

2 for the Japanese style of year, month, day.

Currency display style is given by *co_currstyle* as follows:

0 for the currency symbol to precede the value with no spaces between the symbol and the number.

1 for the currency symbol to follow the value with no spaces between the number and the symbol.

2 for the currency symbol to precede the value with a space after the symbol.

3 for the currency symbol to follow the number with a space before the symbol.

Return Value

On success, *country* returns the pointer argument *cp*. On error, it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

cprintf

See also Example Portability

Syntax

```
#include <conio.h>
int cprintf(const char *format[, argument, ...]);
```

Description

Writes formatted output to the screen.

cprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data directly to the current text window on the screen. There must be the same number of format specifiers as arguments.

For details details on format specifiers, see printf Format Specifiers.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *_directvideo*.

Unlike *fprintf* and *printf*, *cprintf* does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n). Tab characters (specified by \t) are not expanded into spaces.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

cprintf returns the number of characters output.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

cputs

See also Example Portability

Syntax

```
#include <conio.h>
int cputs(const char *str);
```

Description

Writes a string to the screen.

cputs writes the null-terminated string *str* to the current text window. It does not append a newline character.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *_directvideo*. Unlike *puts*, *cputs* does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n).

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

cputs returns the last character printed.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_creat, _wcreat

See also Example Portability

Syntax

```
#include <io.h>
int _creat(const char *path, int amode);
int _wcreat(const wchar_t *path, int amode);
```

Description

Creates a new file or overwrites an existing one.

Note: Remember that a backslash in a path requires '\\'.

_creat creates a new file or prepares to rewrite an existing file given by *path*. *amode* applies only to newly created files.

A file created with creat is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

If the file exists and the write attribute is set, *creat* truncates the file to a length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, the creat call fails and the file remains unchanged.

The *_creat* call examines only the S_IWRITE bit of the access-mode word *amode*. If that bit is 1, the file can be written to. If the bit is 0, the file is marked as read-only. All other operating system attributes are set to 0.

amode can be one of the following (defined in sys\stat.h):

Value of amode	Access permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD / S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successful completion, *_creat* returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following:

EACCES	Permission denied
ENOENT	Path or file name not found
EMFILE	Too many open files

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

creatnew

See also Example Portability

Syntax

```
#include <io.h>
int creatnew(const char *path, int mode);
```

Description

Creates a new file.

creatnew is identical to *_rtl_creat* with one exception: If the file exists, *creatnew* returns an error and leaves the file untouched.

The *mode* argument to *creatnew* can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Return Value

Upon successful completion, *creat* returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EEXIST	File already exists
EMFILE	Too many open files
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

creattemp

See also Example Portability

Syntax

```
#include <io.h>
int creattemp(char *path, int attrib);
```

Description

Creates a unique file in the directory associated with the path name.

A file created with *creattemp* is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

path is a path name ending with a *backslash* (\). A unique file name is selected in the directory given by *path*. The newly created file name is stored in the *path* string supplied. *path* should be long enough to hold the resulting file name. The file is not automatically deleted when the program terminates.

creattemp accepts *attrib*, a DOS attribute word. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

The *attrib* argument to *creattemp* can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Return Value

Upon successful completion, the new file handle, a nonnegative integer, is returned; otherwise, -1 is returned.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_crotl, _crotr

See also Example Portability

Syntax

```
#include <stdlib.h>
unsigned char _crotl(unsigned char val, int count);
unsigned char _crotr(unsigned char val, int count);
```

Description

Rotates an unsigned char left or right.

_crotl rotates the given *val* to the left *count* bits. *_crotr* rotates the given *val* to the right *count* bits.

The argument *val* is an **unsigned char**, or its equivalent in decimal or hexadecimal form.

Return Value

The functions return the rotated word:

- *_crotl* returns the value of *val* left-rotated *count* bits.
- *_crotr* returns the value of *val* right-rotated *count* bits.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

cscanf

See also Example Portability

Syntax

```
#include <conio.h>
int cscanf(char *format[, address, ...]);
```

Description

Scans and formats input from the console.

cscanf scans a series of input fields one character at a time, reading directly from the console. Then each field is formatted according to a format specifier passed to *cscanf* in the format string pointed to by format. Finally, *cscanf* stores the formatted input at an address passed to it as an argument following *format*, and echoes the input directly to the screen. There must be the same number of format specifiers and addresses as there are input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

cscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely for a number of reasons. See *scanf* for a discussion of possible causes.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

cscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *cscanf* attempts to read at end-of-file , the return value is EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

actime, _wctime

See also Example Portability

Syntax

```
#include <time.h>
char *ctime(const time_t *time);
wchar_t *_wctime(const time_t *time);
```

Description

Converts date and time to a string.

ctime converts a time value pointed to by *time* (the value returned by the function *time*) into a 26-character string in the following form, terminating with a newline character and a null character:

```
Mon Nov 21 11:31:54 1983\n\0
```

All the fields have constant width.

The global long variable *_timezone* contains the difference in seconds between GMT and local standard time (in PST, *_timezone* is 8*60*60). The global variable *_daylight* is nonzero *if and only if* the standard U.S. _daylight saving time conversion should be applied. These variables are set by the *tzset* function, not by the user program directly.

Return Value

ctime returns a pointer to the character string containing the date and time. The return value points to static data that is overwritten with each call to *ctime*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ctrlbrk

See also Example Portability

Syntax

```
#include <dos.h>
void ctrlbrk(int (*handler)(void));
```

Description

Sets control-break handler.

ctrlbrk sets a new control-break handler function pointed to by *handler*. The interrupt vector 0x23 is modified to call the named function.

ctrlbrk establishes a DOS interrupt handler that calls the named function; the named function is not called directly.

The handler function can perform any number of operations and system calls. The handler does not have to return; it can use *longjmp* to return to an arbitrary point in the program. The handler function returns 0 to abort the current program; any other value causes the program to resume execution.

Return Value

ctrlbrk returns nothing.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

cwait

See also Example Portability

Syntax

```
#include <process.h>
int cwait(int *statloc, int pid, int action);
```

Description

Waits for child process to terminate.

The *cwait* function waits for a child process to terminate. The process ID of the child to wait for is *pid*. If *statloc* is not NULL, it points to the location where *cwait* will store the termination status. The *action* specifies whether to wait for the process alone, or for the process and all of its children.

If the child process terminated normally (by calling *exit*, or returning from *main*), the termination status word is defined as follows:

Bits 0-7	Zero
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to *exit*, or is returned from *main*. If the child process simply exited from *main* without returning a value, this value will be unpredictable.

If the child process terminated abnormally, the termination status word is defined as follows:

Bits 0-7	Termination information about the child: 1 Critical error abort. 2 Execution fault, protection exception. 3 External termination signal.
Bits 8-15	Zero

If *pid* is 0, *cwait* waits for any child process to terminate. Otherwise, *pid* specifies the process ID of the process to wait for; this value must have been obtained by an earlier call to an asynchronous *spawn* function.

The acceptable values for *action* are WAIT_CHILD, which waits for the specified child only, and WAIT_GRANDCHILD, which waits for the specified child *and* all of its children. These two values are defined in process.h.

Return Value

When *cwait* returns after a normal child process termination, it returns the process ID of the child.

When *cwait* returns after an abnormal child termination, it returns -1 to the parent and sets errno to EINTR (the child process terminated abnormally).

If *cwait* returns without a child process completion, it returns a -1 value and sets *errno* to one of the following values:

ECHILD	No child exists or the pid value is bad
EINVAL	A bad action value was specified

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

delline

See also Example Portability

Syntax

```
#include <conio.h>
void delline(void);
```

Description

Deletes line in text window.

delline deletes the line containing the cursor and moves all lines below it one line up. *delline* operates within the currently active text window.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

difftime

See also Example Portability

Syntax

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Description

Computes the difference between two times.

difftime calculates the elapsed time in seconds, from time1 to time2.

Return Value

difftime returns the result of its calculation as a **double**.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

disable, _disable, enable, _enable

See also Examples Portability

Syntax

```
#include <dos.h>
void disable(void);
void _disable(void);
void enable(void);
void _enable(void);
```

Description

Disables and enables interrupts.

These macros are designed to provide a programmer with flexible hardware interrupt control.

disable and *_disable* macros disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.

enable and *_enable* macros enable interrupts, allowing any device interrupts to occur.

Return Value

None.

Examples

disable

_disable

enable

_enable

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

div

See also Example Portability

Syntax

```
#include <stdlib.h>
div_t  div(int numer, int denom);
```

Description

Divides two integers, returning quotient and remainder.

div divides two integers and returns both the quotient and the remainder as a *div_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *div_t* type is a structure of integers defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
   int  quot;      /* quotient */
   int  rem;       /* remainder */
} div_t;
```

Return Value

div returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

_dos_close

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_close(int handle);
```

Description

Closes a file.

The *_dos_close* function closes the file associated with *handle*; *handle* is a file handle obtained from a _dos_creat, _dos_creatnew, or _dos_open call.

Return Value

Upon successful completion, *_dos_close* returns 0. Otherwise, it returns the operating system error code and the global variable errno is set to

EBADF Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_commit

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_commit(int handle);
```

Description

Outputs a file to the disk.

This function makes DOS flush any output that it has buffered for a specific handle to the disk.

Return Value

The function returns zero on success. On failure the function returns the DOS error code and sets errno to EBADF on failure.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_dos_creat

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_creat(const char *path,int attrib,int *handlep);
```

Description

Creates a new file or overwrites an existing one.

_dos_creat opens the file specified by *path*. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. *_dos_creat* stores the file handle in the location pointed to by *handlep*. The file is opened for both reading and writing.

If the file already exists, its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The *attrib* argument is an ORed combination of one or more of the following constants (defined in dos.h):

_A_NORMAL	Normal file
_A_RDONLY	Read-only file
_A_HIDDEN	Hidden file
_A_SYSTEM	System file

Return Value

On success, *_dos_creat* returns 0.

On error, it returns the operating system error code and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found
EMFILE	Too many open files

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_creatnew

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_creatnew(const char *path, int attrib, int *handlep);
```

Description

Creates a new file.

_dos_creatnew creates and opens the new file *path*. The file is given the access permission *attrib*, an operating-system attribute word. The file is always opened in binary mode. Upon successful file creation, the file handle is stored in the location pointed to by *handlep*, and the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists, *_dos_creatnew* returns an error and leaves the file untouched.

The *attrib* argument to *_dos_creatnew* is an OR combination of one or more of the following constants (defined in dos.h):

_A_NORMAL	Normal file
_A_RDONLY	Read-only file
_A_HIDDEN	Hidden file
_A_SYSTEM	System file

Return Value

Upon successful completion, *_dos_creatnew* returns 0. Otherwise, it returns the operating system error code, and the global variable errno is set to one of the following:

EACCES	Permission denied
EEXIST	File already exists
EMFILE	Too many open files
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

dosexterr

Example Portability

Syntax

```
#include <dos.h>
int dosexterr(struct DOSERROR *eblkp);
```

Description

Gets extended DOS error information.

This function fills in the DOSERROR structure pointed to by *eblkp* with extended error information after a DOS call has failed. The structure is defined as follows:

```
struct  DOSERROR {
   int de_exterror;     /* extended error */
   char de_class;       /* error class */
   char de_action;      /* action */
   char de_locus;       /* error locus */
};
```

The values in this structure are obtained by way of DOS call 0x59. A *de_exterror* value of 0 indicates that the prior DOS call did not result in an error.

Return Value

dosexterr returns the value *de_exterror*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_dos_findfirst

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_findfirst(const char *pathname, int attrib,
                        struct find_t *ffblk);
```

Description

Searches a disk directory.

_dos_findfirst begins a search of a disk directory.

pathname is a string with an optional drive specifier, path, and file name of the file to be found. The file name portion can contain wildcard match characters (such as ? or *). If a matching file is found, the *find_t* structure pointed to by *ffblk* is filled with the file-directory information.

The format of the *find_t* structure is as follows:

```
struct find_t {
   char reserved[21];     /* reserved by the operating system */
   char attrib;           /* attribute found */
   int wr_time;           /* file time */
   int wr_date;           /* file date */
   long size;             /* file size */
   char name[13];         /* found file name */
};
```

attrib is an operating system file-attribute word used in selecting eligible files for the search. *attrib* is an OR combination of one or more of the following constants (defined in dos.h):

_A_NORMAL	Normal file
_A_RDONLY	Read-only attribute
_A_HIDDEN	Hidden file
_A_SYSTEM	System file
_A_VOLID	Volume label
_A_SUBDIR	Directory
_A_ARCH	Archive

For more detailed information about these attributes, refer to your operating system reference manuals.

Note: *wr_time* and *wr_date* contain bit fields for referring to the file's date and time. The structure of these fields was established by the operating system.

wr_time:

Bits 0-4	The result of seconds divided by 2 (for example, 10 here means 20 seconds)
Bits 5-10	Minutes
Bits 11-15	Hours

wr_date:

Bits 0-4	Day
Bits 5-8	Month
Bits 9-15	Years since 1980 (for example, 9 here means 1989)

Return Value

_dos_findfirst returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, the operating system error code is returned, and the global variable errno is set to

ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_findnext

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_findnext(struct find_t *ffblk);
```

Description

Continues *_dos_findfirst* search.

_dos_findnext is used to fetch subsequent files that match the *pathname* given in _dos_findfirst. *ffblk* is the same block filled in by the *_dos_findfirst* call. This block contains necessary information for continuing the search. One file name for each call to *_dos_findnext* is returned until no more files are found in the directory matching the *pathname*.

Return Value

_dos_findnext returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, the operating system error code is returned, and the global variable errno is set to

ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_getdate, _dos_setdate, getdate, setdate

See also Examples Portability

Syntax

```
#include <dos.h>
void _dos_getdate(struct dosdate_t *datep);
unsigned _dos_setdate(struct dosdate_t *datep);
void getdate(struct date *datep);
void setdate(struct date *datep);
```

Description

Gets and sets system date.

getdate fills in the *date* structure (pointed to by *datep*) with the system's current date.

setdate sets the system date (month, day, and year) to that in the *date* structure pointed to by *datep*. Note that a request to set a date might fail if you do not have the privileges required by the operating system.

The *date* structure is defined as follows:

```
struct date{
   int da_year;     /* current year */
   char da_day;     /* day of the month */
   char da_mon;     /* month (1 = Jan) */
};
```

_dos_getdate fills in the *dosdate_t* structure (pointed to by *datep*) with the system's current date.

The *dosdate_t* structure is defined as follows:

```
struct dosdate_t {
   unsigned char day;       /* 1-31 */
   unsigned char month;     /* 1-12 */
   unsigned int  year;      /* 1980 - 2099 */
   unsigned char dayofweek; /* 0 - 6 (0=Sunday) */
};
```

Return Value

_dos_getdate, *getdate*, and *setdate* do not return a value.

If the date is set successfully, *_dos_setdate* returns 0.

Otherwise, it returns a non-zero value and the global variable errno is set to

EINVAL	Invalid date

Examples

getdate

_dos_getdate

_dos_setdate

setdate

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_dos_getdiskfree

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_getdiskfree(unsigned char drive, struct diskfree_t *dtable);
```

Description

Gets disk free space.

_dos_getdiskfree accepts a drive specifier in *drive* (0 for default, 1 for A, 2 for B, and so on) and fills in the *diskfree_t* structure pointed to by *dtable* with disk characteristics.

The *diskfree_t* structure is defined as follows:

```
struct diskfree_t {
   unsigned avail_clusters;       /* available clusters */
   unsigned total_clusters;       /* total clusters */
   unsigned bytes_per_sector;     /* bytes per sector */
   unsigned sectors_per_cluster;  /* sectors per cluster */
};
```

Return Value

_dos_getdiskfree returns 0 if successful. Otherwise, it returns a non-zero value and the global variable errno is set to

EINVAL	Invalid drive specified

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_getdrive, _dos_setdrive

See also Examples Portability

Syntax

```
#include <dos.h>
void _dos_getdrive(unsigned *drivep);
void _dos_setdrive(unsigned drivep, unsigned *ndrives);
```

Description

Gets and sets the current drive number.

_dos_getdrive gets the current drive number.

_dos_setdrive sets the current drive and stores the total number of drives at the location pointed to by ndrives.

The drive numbers at the location pointed to by *drivep* are as follows: 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

Return Value

None. Use *_dos_getdrive* to verify that the current drive was changed successfully.

Examples

_dos_getdrive

_dos_setdrive

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_getfileattr, _dos_setfileattr

See also Examples Portability

Syntax

```
#include <dos.h>
int _dos_getfileattr(const char *path, unsigned *attribp);
int _dos_setfileattr(const char *path, unsigned attrib);
```

Description

Changes file access mode.

_dos_getfileattr fetches the file attributes for the file *path*. The attributes are stored at the location pointed to by *attribp*.

_dos_setfileattr sets the file attributes for the file *path* to the value *attrib*. The file attributes can be an OR combination of the following symbolic constants (defined in dos.h):

_A_RDONLY	Read-only attribute
_A_HIDDEN	Hidden file
_A_SYSTEM	System file
_A_VOLID	Volume label
_A_SUBDIR	Directory
_A_ARCH	Archive
_A_NORMAL	Normal file (no attribute bits set)

Return Value

Upon successful completion, *_dos_getfileattr* and *_dos_setfileattr* return 0. Otherwise, these functions return the operating system error code, and the global variable errno is set to

ENOENT	Path or file name not found

Examples

_dos_getfileattr

_dos_setfileattr

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_getftime, _dos_setftime

See also Examples Portability

Syntax

```
#include <dos.h>
unsigned _dos_getftime(int handle, unsigned *datep, unsigned *timep);
unsigned _dos_setftime(int handle, unsigned date, unsigned time);
```

Description

Gets and sets file date and time.

_dos_getftime retrieves the file time and date for the disk file associated with the open *handle*. The file must have been previously opened using _dos_open, _dos_creat, or _dos_creatnew. *_dos_getftime* stores the date and time at the locations pointed to by *datep* and *timep*.

_dos_setftime sets the file's new date and time values as specified by *date* and *time*.

Note that the date and time values contain bit fields for referring to the file's date and time. The structure of these fields was established by the operating system.

Date:

Bits 0-4	Day
Bits 5-8	Month
Bits 9-15	Years since 1980 (for example, 9 here means 1989)

Time:

Bits 0-4	The result of seconds divided by 2 (for example, 10 here means 20 seconds)
Bits 5-10	Minutes
Bits 11-15	Hours

Return Value

_dos_getftime and *_dos_setftime* return 0 on success.

In the event of an error return, the operating system error code is returned and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Examples

_dos_getftime

_dos_setftime

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_gettime, _dos_settime

See also Examples Portability

Syntax

```
#include <dos.h>
void _dos_gettime(struct dostime_t *timep);
unsigned _dos_settime(struct dostime_t *timep);
```

Description

Gets and sets system time.

_dos_gettime fills in the *dostime_t* structure pointed to by *timep* with the system's current time.

_dos_settime sets the system time to the values in the *dostime_t* structure pointed to by *timep*.

The *dostime_t* structure is defined as follows:

```
struct dostime_t {
   unsigned char hour;     /* hours 0-23 */
   unsigned char minute;   /* minutes 0-59 */
   unsigned char second;   /* seconds 0-59 */
   unsigned char hsecond;  /* hundredths of seconds 0-99 */
};
```

Return Value

_dos_gettime does not return a value.

If *_dos_settime* is successful, it returns 0. Otherwise, it returns the operating system error code, and the global variable errno is set to:

EINVAL	Invalid time

Examples

_dos_gettime

_dos_settime

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_getvect

See also Example Portability

Syntax

```
#include <dos.h>
void interrupt(*_dos_getvect(unsigned interruptno)) ();
```

Description

Gets interrupt vector.

Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

_dos_getvect reads the value of the interrupt vector given by *interruptno* and returns that value as a (far) pointer to an interrupt function. The value of *interruptno* can be from 0 to 255.

Return Value

_dos_getvect returns the current 4-byte value stored in the interrupt vector named by *interruptno*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_dos_open

See also Example Portability

Syntax

```
#include <fcntl.h>
#include <share.h>
#include <dos.h>
unsigned _dos_open(const char *filename, unsigned oflags, int *handlep);
```

Description

Opens a file for reading or writing.

_dos_open open the file specified by *filename*, then prepares it for reading or writing, as determined by the value of *oflags*. The file is always opened in binary mode. *_dos_open* stores the file handle at the location pointed to by *handlep*.

oflags uses the flags from the following two lists. Only one flag from List 1 can be used (and one must be used) and the flags in List 2 can be used in any logical combination.

List 1: Read/write flags

O_RDONLY	Open for reading.
O_WRONLY	Open for writing.
O_RDWR	Open for reading and writing.

The following additional values can be included in *oflags* (using an OR operation):

List 2: Other access flags

O_NOINHERIT	The file is not passed to child programs.
SH_COMPAT	Allow other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode.
SH_DENYRW	Only the current handle can have access to the file.
SH_DENWR	Allow only reads from any other open to the file.
SH_DENYRD	Allow only writes from any other open to the file.
SH_DENYNO	Allow other shared opens to the file, but not other SH_COMPAT opens.

Note: These symbolic constants are defined in fcntl.h and share.h.

Only one of the SH_DENYxx values can be included in a single *_dos_open* routine. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

Return Value

On success:*_dos_open* returns 0 and stores the file handle at the location pointed to by *handlep*. The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, it returns the operating system error code and sets the global variable errno to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	Path or file not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_read

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_read(int handle, void *buf, unsigned len, unsigned *nread);
```

Description

Reads from file.

The *_dos_read* function reads *len* bytes from the file associated with *handle* into the buffer pointed to by the pointer *buf*. The actual number of bytes read is stored at the location pointed to by *nread*; when an error occurs, or the end-of-file is encountered, this number might be less than *len*.

_dos_read does not remove carriage returns because it treats all files as binary files.

handle is a file handle obtained from a _dos_creat, _dos_creatnew, or _dos_open call.

For *_read*, handle is a file handle obtained from a creat, open, dup, or dup2 call.

On disk files, *_dos_read* begins reading at the current file pointer. When the reading is complete, they increment the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that *_dos_read* can read is UINT_MAX -1 (because UINT_MAX is the same as -1, the error return indicator). UINT_MAX is defined in limits.h.

Return Value

On success, *_dos_read* returns 0.

On error, it returns the DOS error code and sets the global variable errno.

On success, *_read* returns a positive integer indicating the number of bytes placed in the buffer. On end-of-file, *_read* returns zero. On error, read returns -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_setvect

See also Example Portability

Syntax

```
#include <dos.h>
void _dos_setvect(unsigned interruptno, void interrupt (*isr) ());
```

Description

Sets interrupt vector entry.

Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

_dos_setvect sets the value of the interrupt vector named by *interruptno* to a new value, isr, which is a far pointer containing the address of a new interrupt function. The address of a C routine can be passed to *isr* only if that routine is declared to be an interrupt routine.

If you use the prototypes declared in dos.h, simply pass the address of an interrupt function to *_dos_setvect* in any memory model.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

dostounix

See also Example Portability

Syntax

```
#include <dos.h>
long dostounix(struct date *d, struct time *t);
```

Description

Converts date and time to UNIX time format.

dostounix converts a date and time as returned from *getdate* and *gettime* into UNIX time format. *d* points to a *date* structure, and *t* points to a *time* structure containing valid date and time information.

The date and time must not be earlier than or equal to Jan 1 1980 00:00:00.

Return Value

UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_dos_write

See also Example Portability

Syntax

```
#include <dos.h>
unsigned _dos_write(int handle, const void far *buf, unsigned len, unsigned
  *nwritten);
unsigned _dos_write(int handle, const void *buf, unsigned len, unsigned
  *nwritten);
```

Description

Writes to a file.

_dos_write writes *len* bytes from the buffer pointed to by pointer *buf* to the file associated with *handle*.

_dos_write does not translate a linefeed character (LF) to a CR/LF pair because it treats all files as binary data.

The actual number of bytes written is stored at the location pointed to by *nwritten*. If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

Return Value

On success, *_dos_write* returns 0.

On error, it returns the operating system error code and sets the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

dup

See also Example Portability

Syntax

```
#include <io.h>
int dup(int handle);
```

Description

Duplicates a file handle.

dup creates a new file handle that has the following in common with the original file handle:

- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

handle is a file handle obtained from a call to *creat, open, dup, dup2, _rtl_creat*, or *_rtl_open*.

Return Value

Upon successful completion, *dup* returns the new file handle, a nonnegative integer; otherwise, *dup* returns -1.

In the event of error, the global variable errno is set to one of the following values:

EBADF	Bad file number
EMFILE	Too many open files

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

dup2

See also Example Portability

Syntax

```
#include <io.h>
int dup2(int oldhandle, int newhandle);
```

Description

Duplicates a file handle (oldhandle) onto an existing file handle (newhandle).

dup2 creates a new file handle that has the following in common with the original file handle:

- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

dup2 creates a new handle with the value of *newhandle*. If the file associated with *newhandle* is open when *dup2* is called, the file is closed.

newhandle and *oldhandle* are file handles obtained from a *creat, open, dup,* or *dup2* call.

Return Value

dup2 returns 0 on successful completion, -1 otherwise.

In the event of error, the global variable errno is set to one of the following values:

EBADF Bad file number

EMFILE Too many open files

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

ecvt

See also Example Portability

Syntax

```
#include <stdlib.h>
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

ecvt converts *value* to a null-terminated string of *ndig* digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative, the word pointed to by *sign* is nonzero; otherwise, it's 0. The low-order digit is rounded.

Return Value

The return value of *ecvt* points to static data for the string of digits whose content is overwritten by each call to *ecvt* and *fcvt*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_ _emit_ _

Example Portability

Syntax

```
#include <dos.h>
void _ _emit_ _(argument, ...);
```

Description

Inserts literal values directly into code.

_ _*emit*_ _ is an inline function that lets you insert literal values directly into object code as it is compiling. It is used to generate machine language instructions without using inline assembly language or an assembler.

Generally the arguments of an _ _*emit*_ _ call are single-byte machine instructions. However, because of the capabilities of this function, more complex instructions, complete with references to C variables, can be constructed.

You should use this function only if you are familiar with the machine language of the 80x86 processor family. You can use this function to place arbitrary bytes in the instruction code of a function; if any of these bytes is incorrect, the program misbehaves and can easily crash your machine. Borland C++ does not attempt to analyze your calls for correctness in any way. If you encode instructions that change machine registers or memory, Borland C++ will not be aware of it and might not properly preserve registers, as it would in many cases with inline assembly language (for example, it recognizes the usage of SI and DI registers in inline instructions). You are completely on your own with this function.

You must pass at least one argument to _ _*emit*_ _; any number can be given. The arguments to this function are not treated like any other function call arguments in the language. An argument passed to _ _*emit*_ _ will not be converted in any way.

There are special restrictions on the form of the arguments to _ _*emit*_ _. Arguments must be in the form of expressions that can be used to initialize a static object. This means that integer and floating-point constants and the addresses of static objects can be used. The values of such expressions are written to the object code at the point of the call, exactly as if they were being used to initialize data. The address of a parameter or auto variable, plus or minus a constant offset, can also be used. For these arguments, the offset of the variable from BP is stored.

The number of bytes placed in the object code is determined from the type of the argument, except in the following cases:

- If a signed integer constant (that is 0x90) appears that fits within the range of 0 to 255, it is treated as if it were a character.
- If the address of an auto or parameter variable is used, a byte is written if the offset of the variable from BP is between -128 and 127; otherwise, a word is written.

Simple bytes are written as follows:

```
_ _emit_ _(0x90);
```

If you want a word written, but the value you are passing is under 255, simply cast it to **unsigned** using one of these methods:

```
_ _emit_ _(0xB8, (unsigned)17);
_ _emit_ _(0xB8, 17u);
```

Two- or four-byte address values can be forced by casting an address to **void near** * or **void far** *, respectively.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_endthread

See also Examples Portability

Syntax

```
#include <process.h>
void _endthread(void);
```

Description

Terminates execution of a thread.

The *_endthread* function terminates the currently executing thread. The thread must have been started by an earlier call to _beginthread or _beginthreadNT.

This function is available in the multithread libraries; it is not in the single-thread libraries.

Return Value

The function does not return a value.

Examples

_beginthread (Win32s version)

_beginthreadNT (Windows NT version)

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

eof

See also Example Portability

Syntax

```
#include <io.h>
int eof(int handle);
```

Description

Checks for end-of-file.

eof determines whether the file associated with *handle* has reached end-of-file.

Return Value

If the current position is end-of-file, *eof* returns the value 1; otherwise, it returns 0. A return value of -1 indicates an error; the global variable errno is set to

EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

execl, execle, execlp, execlpe, execv, execve, execvp, execvpe

See also Examples Portability

Syntax

```
#include <process.h>
int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);
int _wexecl(wchar_t *path, wchar_t *arg0 *arg1, ..., *argn, NULL);
int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexecle(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t
  **env);


int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);
int _wexeclp(wchar_t *path, wchar_t *arg0,*arg1, ..., *argn, NULL);
int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexeclpe(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t
  **env);

int execv(char *path, char *argv[]);
int _wexecv(wchar_t *path, wchar_t *argv[]);
int execve(char *path, char *argv[], char **env);
int _wexecve(wchar_t *path, wchar_t *argv[], wchar_t **env);

int execvp(char *path, char *argv[]);
int _wexecvp(wchar_t *path, wchar_t *argv[]);
int execvpe(char *path, char *argv[], char **env);
int _wexecvpe(wchar_t *path, wchar_t *argv[], wchar_t **env);
```

Description

Loads and runs other programs.

The functions in the *exec...* family load and run (execute) other programs, known as *child processes*. When an *exec...* call succeeds, the child process overlays the *parent process*. There must be sufficient memory available for loading and executing the child process.

path is the file name of the called child process. The *exec...* functions search for *path* using the standard search algorithm:

- If no explicit extension is given, the functions search for the file as given. If the file is not found, they add .EXE and search again. If not found, they add .COM and search again. If found, the command processor, COMSPEC (Windows) or COMMAND.COM (DOS), is used to run the batch file.
- If an explicit extension or a period is given, the functions search for the file exactly as given.

The suffixes *l*, *v*, *p*, and *e* added to the *exec...* "family name" specify that the named function operates with certain capabilities.

l specifies that the argument pointers (*arg0, arg1, ..., argn*) are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.

v specifies that the argument pointers (*argv[0] ..., arg[n]*) are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.

p specifies that the function searches for the file in those directories specified by the PATH environment variable (without the *p* suffix, the function searches only the current working directory). If the *path* parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.

e specifies that the argument *env* can be passed to the child process, letting you alter the environment for the child process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the *exec...* family must have one of the two argument-specifying suffixes (either *l* or *v*).

The path search and environment inheritance suffixes *(p* and *e*) are optional; for example:

- *execl* is an *exec*... function that takes separate arguments, searches only the root or current directory for the child, and passes on the parent's environment to the child.
- *execvpe* is an *exec*... function that takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *env* argument for altering the child's environment.

The *exec...* functions must pass at least one argument to the child process (*arg0 or argv[0]*); this argument is, by convention, a copy of *path*. (Using a different value for this 0th argument won't produce an error.)

path is available for the child process.

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1*, *..., argn* point to character strings that form the new list of arguments. A mandatory null following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *env*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *env* is null. When *env* is null, the child inherits the parents' environment settings.

The combined length of *arg0* + *arg1* + ... + *argn* (or of *argv[0]* + *argv[1]* + ... + *argn[n]*), including space characters that separate the arguments, must be less than 128 bytes for a 16-bit application, or 260 bytes for Win32 application. Null terminators are not counted.

When an *exec...* function call is made, any open files remain open in the child process.

Return Value

If successful, the *exec...* functions do not return. On error, the *exec...* functions return -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Examples

execl
execle
execlp
execlpe
execv
execve
execvp
execvpe

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_exit

See also Example Portability

Syntax

```
#include <stdlib.h>
void _exit(int status);
```

Description

Terminates program.

_exit terminates execution without closing any files, flushing any output, or calling any exit functions.

The calling process uses *status* as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

exit

See also Example Portability

Syntax

```
#include <stdlib.h>
void exit(int status);
```

Description

Terminates program.

exit terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with *atexit*) are called.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:

EXIT_FAILURE	Abnormal program termination; signal to operating system that program has terminated with an error
EXIT_SUCCESS	Normal program termination

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

exp, expl

See also Example Portability

Syntax

```
#include <math.h>
double exp(double x);
long double expl(long double x);
```

Description

Calculates the exponential *e* to the *x*.

expl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with bcd and complex types.

Return Value

exp returns *e* to the *x*.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value overflows, *exp* returns the value HUGE_VAL and *expl* returns _LHUGE_VAL. Results of excessively large magnitude cause the global variable errno to be set to

ERANGE	Result out of range

On underflow, these functions return 0.0, and the global variable errno is not changed. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
exp	+	+	+	+	+	+	+
expl	+		+	+			+

_expand

See also Example Portability

Syntax

```
#include <malloc.h>
void *_expand(void *block, size_t size);
```

Description

Grows or shrinks a heap block in place.

This function attempts to change the size of an allocated memory *block* without moving the block's location in the heap. The data in the *block* are not changed, up to the smaller of the old and new sizes of the block. The block must have been allocated earlier with *malloc*, *calloc*, or *realloc*, and must not have been freed.

Return Value

If *_expand* is able to resize the block without moving it, *_expand* returns a pointer to the block, whose address is unchanged. If *_expand* is unsuccessful, it returns a NULL pointer and does not modify or resize the block.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

fabs, fabsl

See also Example Portability

Syntax

```
#include <math.h>
double fabs(double x);
long double fabsl(long double x);
```

Description

Returns the absolute value of a floating-point number.

fabs calculates the absolute value of *x*, a double. *fabsl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

fabs and *fabsl* return the absolute value of *x*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
fabs	+	+	+	+	+	+	+
fabsl	+		+	+			+

farcalloc

See also Example Portability

Syntax

```
#include <alloc.h>
void far *farcalloc(unsigned long nunits, unsigned long unitsz);
```

Description

Allocates memory from the far heap.

farcalloc allocates memory from the far heap for an array containing *nunits* elements, each *unitsz* bytes long.

For allocating from the far heap, note that:

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers (or huge pointers if blocks are larger than 64K) are used to access the allocated blocks.

In the compact, large, and huge memory models, farcalloc is similar, though not identical, to *calloc*. It takes **unsigned long** parameters, while *calloc* takes **unsigned** parameters. For DOS users, a tiny model program cannot use *faralloc.*

Return Value

farcalloc returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

farfree

See also Example Portability

Syntax

```
#include <alloc.h>
void farfree(void far * block);
```

Description

Frees a block from far heap.

farfree releases a block of memory previously allocated from the far heap.

In the small and medium memory models, blocks allocated by *farmalloc* cannot be freed with normal *free*, and blocks allocated with *malloc* cannot be freed with *farfree*. In these models, the two heaps are completely distinct. For DOS users, a tiny model program cannot use *farfree.*

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

farmalloc

See also Example Portability

Syntax

```
#include <alloc.h>
void far *farmalloc(unsigned long nbytes);
```

Description

Allocates from far heap.

farmalloc allocates a block of memory *nbytes* bytes long from the far heap.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

In the compact, large, and huge memory models, *farmalloc* is similar though not identical to *malloc*. It takes **unsigned long** parameters, while *malloc* takes **unsigned** parameters. For DOS users, a tiny model program cannot use *farmalloc.*

Return Value

farmalloc returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

farrealloc

See also Example Portability

Syntax

```
#include <alloc.h>
void far *farrealloc(void far *oldblock, unsigned long nbytes);
```

Description

Adjusts allocated block in far heap.

farrealloc adjusts the size of the allocated block to *nbytes* copying the contents to a new location if necessary.

For allocating from the far heap:

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

For DOS users, a tiny model program cannot use *farrealloc.*

Return Value

farrealloc returns the address of the reallocated block which might be different than the address of the original block. If the block cannot be reallocated *farrealloc* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

fclose

See also Example Portability

Syntax

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

Closes a stream.

fclose closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with *setbuf* or *setvbuf* are not automatically freed. (But if *setvbuf* is passed null for the buffer pointer it *will* free it upon close.)

Return Value

fclose returns 0 on success. It returns EOF if any errors were detected.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fcloseall

See also Example Portability

Syntax

```
#include <stdio.h>
int fcloseall(void);
```

Description

Closes open streams.

fcloseall closes all open streams except

stdin

stdout

stdprn

stderr

stdauxstdstreams

Note: stdprn and stdaux streams are not available in OS/2 and Win32.

Return Value

fcloseall returns the total number of streams it closed. It returns EOF if any errors were detected.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

fcvt

See also Example Portability

Syntax

```
#include <stdlib.h>
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

fcvt converts *value* to a null-terminated string digit starting with the leftmost significant digit with *ndig* digits to the right of the decimal point. *fcvt* then returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative the word pointed to by *sign* is nonzero; otherwise it is 0.

The correct digit has been rounded for the number of digits to the right of the decimal point specified by *ndig*.

Return Value

The return value of *fcvt* points to static data whose content is overwritten by each call to *fcvt* and *ecvt*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_fdopen, _wfdopen

See also Example Portability

Syntax

```
#include <stdio.h>
FILE *_fdopen(int handle, char *type);
FILE *_wfdopen(int handle, wchar_t *type);
```

Description

Associates a stream with a file handle.

_fdopen associates a stream with a file handle obtained from creat, dup, dup2, or open.

The type of stream must match the mode of the open *handle*.

The *type* string used in a call to *_fdopen* is one of the following values:

Value	Description
r	Open for reading only. *_fdopen* returns NULL if the file cannot be opened.
w	Create for writing. If the file already exists, its contents are overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing). *_fdopen* returns NULL if the file cannot be opened.
w+	Create a new file for update. If the file already exists, its contents are overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, append *t* to the value of the *type* string (for example, `rt` or `w+t`).

Similarly, to specify binary mode append *b* to the *type* string (for example, `rb` or `w+b`).

If *t* or *b* is not given in the type string, the mode is governed by the global variable _fmode.

If *_fmode* is set to O_BINARY, files will be opened in binary mode.

If *_fmode* is set to O_TEXT, files will be opened in text mode.

Note: The O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseekor rewind
- input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-offile

Return Value

On successful completion *_fdopen* returns a pointer to the newly opened stream. In the event of error it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

feof

See also Example Portability

Syntax

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

Detects end-of-file on a stream.

feof is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set read operations on the file return the indicator until *rewind* is called or the file is closed. The end-of-file indicator is reset with each input operation.

Return Value

feof returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ferror

See also Example Portability

Syntax

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

Detects errors on stream.

ferror is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set it remains set until *clearerr* or *rewind* is called or until the stream is closed.

Return Value

ferror returns nonzero if an error was detected on the named stream.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fflush

See also Example Portability

Syntax

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

Flushes a stream.

If the given stream has buffered output *fflush* writes the output for *stream* to the associated file.

The stream remains open after *fflush* has executed. *fflush* has no effect on an unbuffered stream.

Return Value

fflush returns 0 on success. It returns EOF if any errors were detected.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fgetc

See also Example Portability

Syntax

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

Gets character from stream.

fgetc returns the next character on the named input stream.

Return Value

On success *fgetc* returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_fgetchar, _fgetwchar

See also Example Portability

Syntax

```
#include <stdio.h>
int _fgetchar(void);
wint_t _fgetwchar(void);
```

Description

Reads a character from stdin.

_fgetchar returns the next character from stdin. It is defined as fgetc(stdin).

Note: For Win32s or Win32 GUI applications, stdin must be redirected.

Return Value

On success *_fgetchar* returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

fgetpos

See also Example Portability

Syntax

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

Gets the current file pointer.

fgetpos stores the position of the file pointer associated with the given stream in the location pointed to by *pos*. The exact value is unimportant; its value is opaque except as a parameter to subsequent *fsetpos* calls.

Return Value

On success *fgetpos* returns 0. On failure it returns a nonzero value and sets the global variable *errno* to

EBADF	Bad file number
EINVAL	Invalid number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

fgets, fgetws

See also Example Portability

Syntax

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream); // Unicode version
```

Description

Gets a string from a stream.

fgets reads characters from *stream* into the string *s*. The function stops reading when it reads either *n* - 1 characters or a newline character whichever comes first. *fgets* retains the newline character at the end of *s*. A null byte is appended to s to mark the end of the string.

Return Value

On success *fgets* returns the string pointed to by *s*; it returns NULL on end-of-file or error.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

filelength

See also Example Portability

Syntax

```
#include <io.h>
long filelength(int handle);
```

Description

Gets file size in bytes.

filelength returns the length (in bytes) of the file associated with *handle*.

Return Value

On success *filelength* returns a **long** value the file length in bytes. On error it returns -1 and the global variable *errno* is set to

EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fileno

See also Example Portability

Syntax

```
#include <stdio.h>
int fileno(FILE *stream);
```

Description

Gets file handle.

fileno is a macro that returns the file handle for the given stream. If *stream* has more than one handle *fileno* returns the handle assigned to the stream when it was first opened.

Return Value

fileno returns the integer file handle associated with *stream*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_findfirst, _wfindfirst

See also Example Portability

Syntax

```
#include <dir.h>
int _findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
int _wfindfirst(const wchar_t *pathname, struct _wffblk *ffblk, int attrib);
```

Description

Searches a disk directory.

_findfirst begins a search of a disk directory for files specifed by attributes or wildcards.

pathname is a string with an optional drive specifier path and file name of the file to be found. Only the file name portion can contain wildcard match characters (such as ? or *). If a matching file is found the *ffblk* structure is filled with the file-directory information.

When Unicode is defined, the*_wfindfirst* function uses the following *_wffblk* structure.

```
struct  _wffblk   {
    long            ff_reserved;
    long            ff_fsize;
    unsigned long   ff_attrib;
    unsigned short  ff_ftime;
    unsigned short  ff_fdate;
    wchar_t         ff_name[256];
};
```

Win16

For Win16, the format of the structure ffblk is as follows:

```
struct ffblk {
   char ff_reserved[21];     /* reserved by DOS */
   char ff_attrib;           /* attribute found */
   int ff_ftime;             /* file time */
   int ff_fdate;             /* file date */
   long ff_fsize;            /* file size */
   char ff_name[13];         /* found file name */
   };
```

Win32

For Win32, the format of the structure *ffblk* is as follows:

```
struct ffblk {
   long           ff_reserved;
   long           ff_fsize;       /* file size */
   unsigned long  ff_attrib;      /* attribute found */
   unsigned short ff_ftime;       /* file time */
   unsigned short ff_fdate;       /* file date */
   char           ff_name[256];   /* found file name */
   };
```

attrib is a file-attribute byte used in selecting eligible files for the search. *attrib* should be selected from the following constants defined in dos.h:

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory

FA_ARCH	Archive

A combination of constants can be ORed together.

For more detailed information about these attributes refer to your operating system documentation.

ff_ftime and *ff_fdate* contain bit fields for referring to the current date and time. The structure of these fields was established by the operating system. Both are 16-bit structures divided into three fields.

ff_ftime:

Bits 0 to 4	The result of seconds divided by 2 (for example 10 here means 20 seconds)
Bits 5 to 10	Minutes
Bits 11 to 15	Hours

ff_fdate:

Bits 0-4	Day
Bits 5-8	Month
Bits 9-15	Years since 1980 (for example 9 here means 1989)

The structure *ftime* declared in io.h uses time and date bit fields similar in structure to *ff_ftime* and *ff_fdate*.

Return Value

_findfirst returns 0 on successfully finding a file matching the search *pathname*.

When no more files can be found, or if there is an error in the file name:

- -1 is returned
- errno is set to

  ENOENT Path or file name not found
- _doserrno is set to one of the following values:

  ENMFILE No more files

  ENOENT Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_findnext, _wfindnext

See also Example Portability

Syntax

```
#include <dir.h>
int _findnext(struct ffblk *ffblk);
int _wfindnext(struct _wffblk *ffblk);
```

Description

Continues _findfirst search.

_findnext is used to fetch subsequent files that match the *pathname* given in *findfirst. ffblk* is the same block filled in by the *findfirst* call. This block contains necessary information for continuing the search. One file name for each call to *_findnext* will be returned until no more files are found in the directory matching the *pathname*.

Return Value

_findnext returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found or if there is an error in the file name

-1 is returned

errno is set to

ENOENT Path or file name not found

_doserrno is set to one of the following values:

ENMFILE No more files

ENOENT Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

floor, floorl

See also Example Portability

Syntax

```
#include <math.h>
double floor(double x);
long double floorl(long double x);
```

Description

Rounds down.

floor finds the largest integer not greater than *x*.

floorl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

floor returns the integer found as a **double**. *floorl* returns the integer found as a **long double**.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
floor	+	+	+	+	+	+	+
floorl	+		+	+			+

flushall

See also Example Portability

Syntax

```
#include <stdio.h>
int flushall(void);
```

Description

Flushes all streams.

flushall clears all buffers associated with open input streams and writes all buffers associated with open output streams to their respective files. Any read operation following *flushall* reads new data into the buffers from the input files. Streams stay open after *flushall* executes.

Return Value

flushall returns an integer the number of open input and output streams.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

fmod, fmodl

See also Example Portability

Syntax

```
#include <math.h>
double fmod(double x, double y);
long double fmodl(long double x, long double y);
```

Description

Calculates *x* modulo *y*, the remainder of *x/y*.

fmod calculates *x* modulo *y* (the remainder *f*, where $x = ay + f$ for some integer *a*, and $0 \leq f < y$).

fmodl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

fmod and *fmodl* return the remainder *f* where $x = ay + f$ (as described above). When y = 0, *fmod* and *fmodl* return 0.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
fmod	+	+	+	+	+	+	+
fmodl	+		+	+			+

fnmerge, _wfnmerge

See also Example Portability

Syntax

```
#include <dir.h>
void fnmerge(char *path, const char *drive, const char *dir, const char
  *name, const char *ext);
void _wfnmerge(wchar_t *path, const wchar_t *drive, const wchar_t *dir,
  const wchar_t *name, const wchar_t *ext );
```

Description

Builds a path from component parts.

fnmerge makes a path name from its components. The new path name is

`X:\DIR\SUBDIR\NAME.EXT`

where:

drive	=	X
dir	=	\\DIR\\SUBDIR\\
name	=	NAME
ext	=	.EXT

If *drive* is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If *dir* is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If *name* is empty or NULL, no file name is inserted in the path name.

If *ext* is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

fnmerge assumes there is enough space in *path* for the constructed path name. The maximum constructed length is MAXPATH. MAXPATH is defined in dir.h.

fnmerge and *fnsplit* are invertible; if you split a given *path* with *fnsplit* then merge the resultant components with *fnmerge* you end up with *path*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fnsplit, _wfnsplit

See also Example Portability

Syntax

```
#include <dir.h>
int fnsplit(const char *path, char *drive, char *dir, char *name, char
  *ext);
int _wfnsplit(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t
  *name, wchar_t *ext );
```

Description

Splits a full path name into its components.

fnsplit takes a file's full path name *(path)* as a string in the form X:\DIR\SUBDIR\NAME.EXT and splits *path* into its four components. It then stores those components in the strings pointed to by *drive, dir, name*, and *ext*. All five components must be passed but any of them can be a null which means the corresponding component will be parsed but not stored. If any path component is null, that component corresponds to a non-NULL, empty string.

The maximum sizes for these strings are given by the constants *MAXDRIVE, MAXDIR, MAXPATH, MAXFILE*, and *MAXEXT* (defined in dir.h) and each size includes space for the null-terminator.

Constant	Max 16-bit	Max 32-bit	String
MAXPATH	80	256	path
MAXDRIVE	3	3	drive; includes colon (:)
MAXDIR	66	260	dir; includes leading and trailing backslashes (\)
MAXFILE	9	256	name
MAXEXT	5	256	ext; includes leading dot (.)

fnsplit assumes that there is enough space to store each non-null component.

When *fnsplit* splits *path* it treats the punctuation as follows:

- *drive* includes the colon (C:, A:, and so on)
- *dir* includes the leading and trailing backslashes (\BC\include\, \source\ ,and so on)
- *name* includes the file name
- *ext* includes the dot preceding the extension (.C, .EXE, and so on).

fnmerge and *fnsplit* are invertible; if you split a given *path* with *fnsplit* then merge the resultant components with *fnmerge* you end up with *path*.

Return Value

fnsplit returns an integer (composed of five flags defined in dir.h) indicating which of the full path name components were present in *path*. These flags and the components they represent are

EXTENSION	An extension
FILENAME	A file name
DIRECTORY	A directory (and possibly subdirectories)
DRIVE	A drive specification (see dir.h)
WILDCARDS	Wildcards (* or ?)

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fopen, _wfopen

See also Example Portability

Syntax

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
FILE *_wfopen(const wchar_t *filename, const wchar_t *mode);
```

Description

Opens a stream.

fopen opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

The *mode* string used in calls to *fopen* is one of the following values:

Value	Description
r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a *t* to the *mode* string (*rt w+t* and so on). Similarly to specify binary mode append a *b* to the *mode* string (*wb a+b* and so on). *fopen* also allows the *t* or *b* to be inserted between the letter and the + character in the mode string; for example *rt+* is equivalent to *r+t*.

If *a t* or *b* is not given in the *mode* string the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY files are opened in binary mode. If *_fmode* is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-offile

Return Value

On successful completion *fopen* returns a pointer to the newly opened stream. In the event of error it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

FP_OFF, FP_SEG

See also Example Portability

Syntax

```
#include <dos.h>
unsigned FP_OFF(void far *p);
unsigned FP_SEG(void far *p);
```

Description

Gets a far address offset or segment.

FP_OFF is a macro that gets or sets the offset of the **far** pointer *p*.

FP_SEG is a macro that gets or sets the segment value of the **far** pointer *p*.

Return Value

FP_OFF returns an **unsigned** integer value representing an offset value.

FP_SEG returns an **unsigned** integer representing a segment value.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_fpreset

See also Portability

Syntax

```
#include <float.h>
void _fpreset(void);
```

Description

Reinitializes floating-point math package.

_fpreset reinitializes the floating-point math package. This function is usually used in conjunction with *system* or the *exec...* or *spawn...* functions. It is also used to recover from floating-point errors before calling *longjmp*.

Note: If an 80x87 coprocessor is used in a program a child process (executed by the system, or by an exec... or spawn... function) might alter the parent process' floating-point state.

If you use an 80x87 take the following precautions:

- Do not call *system* or an *exec...* or *spawn...* function while a floating-point expression is being evaluated.
- Call *_fpreset* to reset the floating-point state after using *system exec...* or *spawn...* if there is *any* chance that the child process performed a floating-point operation with the 80x87.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fprintf, fwprintf

See also Example Portability

Syntax

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format[, argument, ...]);
int fwprintf(FILE *stream, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a stream.

fprintf accepts a series of arguments applies to each a format specifier contained in the format string pointed to by *format* and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Note: For details on format specifiers, see printf Format Specifiers.

Return Value

fprintf returns the number of bytes output. In the event of error it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fputc, fputwc

See also Example Portability

Syntax

```
#include <stdio.h>
int fputc(int c, FILE *stream);
wint_t fputwc(wint_t c, FILE *stream);
```

Description

Puts a character on a stream.

fputc outputs character c to the named stream.

Note: For Win32s or Win32 GUI applications, stdin must be redirected.

Return Value

On success, *fputc* returns the character *c*. On error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_fputchar, _fputwchar

See also Example Portability

Syntax

```
#include <stdio.h>
int _fputchar(int c);
wint_t _fputwchar(wint_t c);
```

Description

Outputs a character to stdout.

_fputchar outputs character *c* to stdout. *_fputchar(c)* is the same as *fputc(cstdout)*.

For Win32s or Win32 GUI applications, stdout must be redirected.

Return Value

On success *_fputchar* returns the character *c*.

On error it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+		+

fputs, fputws

See also Example Portability

Syntax

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
```

Description

Outputs a string on a stream.

fputs copies the null-terminated string *s* to the given output stream; it does not append a newline character and the terminating null character is not copied.

Return Value

On success *fputs* returns a non-negative value.

On error it returns a value of EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fread

See also Example Portability

Syntax

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Reads data from a stream.

fread reads *n* items of data each of length *size* bytes from the given input stream into a block pointed to by *ptr*.

The total number of bytes read is *(n * size)*.

Return Value

On success *fread* returns the number of items (not bytes) actually read.

On end-of-file or error it returns a short count (possibly 0).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

free

See also Example Portability

Syntax

```
#include <stdlib.h>
void free(void *block);
```

Description

Frees allocated block.

free deallocates a memory block allocated by a previous call to *calloc, malloc,* or *realloc*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

freopen, _wfreopen

See also Example Portability

Syntax

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
FILE *_wfreopen(const wchar_t *filename, const wchar_t *mode, FILE *stream);
```

Description

Associates a new file with an open stream.

freopen substitutes the named file in place of the open stream. It closes *stream* regardless of whether the open succeeds. *freopen* is useful for changing the file attached to stdin, stdout, or stderr.

The *mode* string used in calls to *fopen* is one of the following values:

Value	Description
r	Open for reading only.
w	Create for writing. .
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing).
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a *t* to the *mode* string (*rt w+t* and so on); similarly to specify binary mode append a *b* to the *mode* string (*wb a+b* and so on).

If a *t* or *b* is not given in the *mode* string the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY files are opened in binary mode. If *_fmode* is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseekor rewind
- input cannot be directly followed by output without an intervening *fseek, rewind,* or an input that encounters end-offile

Return Value

On successful completion *freopen* returns the argument *stream*.

On error it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

frexp, frexpl

See also Example Portability

Syntax

```
#include <math.h>
double frexp(double x, int *exponent);
long double frexpl(long double x, int *exponent);
```

Description

Splits a number into mantissa and exponent.

frexp calculates the mantissa *m* (a **double** greater than or equal to 0.5 and less than 1) and the integer value *n* such that *x* (the original **double** value) equals *m * 2n. frexp* stores *n* in the integer that *exponent* points to.

frexpl is the **long double** version; it takes a **long double** argument for *x* and returns a **long double** result.

Return Value

frexp and *frexpl* return the mantissa *m*. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
frexp	+	+	+	+	+	+	+
frexpl	+		+	+			+

fscanf, fwscanf

See also Example Portability

Syntax

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format[, address, ...]);
int fwscanf(FILE *stream, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a stream.

fscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to *fscanf* in the format string pointed to by format. Finally *fscanf* stores the formatted input at an address passed to it as an argument following *format*. The number of format specifiers and addresses must be the same as the number of input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

fscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See *scanf* for a discussion of possible causes.

Return Value

fscanf returns the number of input fields successfully scanned, converted and stored. The return value does not include scanned fields that were not stored.

If *fscanf* attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fseek

See also Example Portability

Syntax

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Description

Repositions a file pointer on a stream.

fseek sets the file pointer associated with *stream* to a new position that is *offset* bytes from the file location given by *whence*. For text mode streams offset should be 0 or a value returned by *ftell*.

whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows:

Constant	whence	File location
SEEK_SET	0	File beginning
SEEK_CUR	1	Current file pointer position
SEEK_END	2	End-of-file

fseek discards any character pushed back using *ungetc*. *fseek* is used with stream I/O; for file handle I/O use *lseek*.

After *fseek* the next operation on an update file can be either input or output.

Return Value

fseek returns 0 if the pointer is successfully moved and nonzero on failure.

fseek might return a 0 indicating that the pointer has been moved successfully when in fact it has not been. This is because DOS, which actually resets the pointer, does not verify the setting. *fseek* returns an error code only on an unopened file or device.

In the event of an error return the global variable *errno* is set to one of the following values:

EBADF	Bad file pointer
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fsetpos

See also Example Portability

Syntax

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

Positions the file pointer of a stream.

fsetpos sets the file pointer associated with *stream* to a new position. The new position is the value obtained by a previous call to *fgetpos* on that stream. It also clears the end-of-file indicator on the file that *stream* points to and undoes any effects of *ungetc* on that file. After a call to *fsetpos* the next operation on the file can be input or output.

Return Value

On success *fsetpos* returns 0.

On failure it returns a nonzero value and also sets the global variable errno to a nonzero value.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

_fsopen

See also Example Portability

Syntax

```
#include <stdio.h>
#include <share.h>
FILE *_fsopen(const char *filename, const char *mode, int shflag);
```

Description

Opens a stream with file sharing.

_fsopen opens the file named by *filename* and associates a stream with it. *_fsopen* returns a pointer that is used to identify the stream in subsequent operations.

The *mode* string used in calls to *_fsopen* is one of the following values:

Mode	Description
r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end of file. or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a *t* to the *mode* string (*rt w+t* and so on). Similarly to specify binary mode append a *b* to the *mode* string (*wb a+b* and so on). *_fsopen* also allows the *t* or *b* to be inserted between the letter and the + character in the mode string; for example *rt+* is equivalent to *r+t*. If a *t* or *b* is not given in the *mode* string the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY files are opened in binary mode. If *_fmode* is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream, however:

- output cannot be directly followed by input without an intervening fseekor rewind
- input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-offile

shflag specifies the type of file-sharing allowed on the file *filename*. Symbolic constants for sh*fl*ag are defined in share.h.

Note: For DOS users, the file-sharing flags are ignored if SHARE is not loaded.

Value of shflag	Description
SH_COMPAT	Sets compatibility mode
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access
SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On successful completion *_fsopen* returns a pointer to the newly opened stream.

On error it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fstat, stat, _wstat

See also Examples Portability

Syntax

```
#include <sys\stat.h>
int fstat(int handle, struct stat *statbuf);
int stat(const char *path, struct stat *statbuf);
int _wstat(const wchar_t *path, struct stat *statbuf);
```

Description

Gets open file information.

fstat stores information in the *stat* structure about the file or directory associated with *handle*.

stat stores information about a given file or directory in the *stat* structure. The name of the file is *path*.

statbuf points to the *stat* structure (defined in sys\stat.h). That structure contains the following fields:

st_mode	Bit mask giving information about the file's mode
st_dev	Drive number of disk containing the file or file handle if the file is on a device
st_rdev	Same as *st_dev*
st_nlink	Set to the integer constant 1
st_size	Size of the file in bytes
st_atime	Most recent access (Windows) or last time modified (DOS)
st_mtime	Same as *st_atime*
st_ctime	Same as *st_atime*

The *stat* structure contains three more fields not mentioned here. They contain values that are meaningful only in UNIX.

The *st_mode* bit mask that gives information about the mode of the open file includes the following bits:

One of the following bits will be set:

S_IFCHR	If *handle* refers to a device.
S_IFREG	If an ordinary file is referred to by *handle*.

One or both of the following bits will be set:

S_IWRITE	If user has permission to write to file.
S_IREAD	If user has permission to read to file.

The HPFS and NTFS file-management systems make the following distinctions:

st_atime	Most recent access
st_mtime	Most recent modify
st_ctime	Creation time

Return Value

fstat and *stat* return 0 if they successfully retrieved the information about the open file.

On error (failure to get the information) these functions return -1 and set the global variable *errno* to

EBADF	Bad file handle

Examples

fstat

stat

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_fstr*

See also Portability

Syntax

```
#include <string.h>
__far string functions
```

Description

Provides string operations in a large-code model.

Choose **See Also** to see a list of string functions that have a **__far** version. The **__far** version of a string function is prefixed with *_fstr*. The behavior of a **__far** string function is identical to the behavior of the standard function to which it corresponds. The only difference is that arguments and return value (if any) to a **__far** string function are modified by the **__far** keyword. The entry for each of the functions provides a description for the **__far** version.

Return Value

The return value for a *_fstr*-type function is a **__far** type.

Note: When a far string function returns an **int** or *size_t*, the return value is never modified by the **far** keyword.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

ftell

See also Example Portability

Syntax

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

Returns the current file pointer.

ftell returns the current file pointer for *stream*. The offset is measured in bytes from the beginning of the file (if the file is binary). The value returned by *ftell* can be used in a subsequent call to *fseek*.

Return Value

ftell returns the current file pointer position on success. It returns -1L on error and sets the global variable *errno* to a positive value.

In the event of an error return the global variable *errno* is set to one of the following values:

EBADF	Bad file pointer
ESPIPE	Illegal seek on device

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ftime

See also Example Portability

Syntax

```
#include <sys\timeb.h>
void ftime(struct timeb *buf)
```

Description

Stores current time in timeb structure.

On UNIX platforms *ftime* is available only on System V systems.

ftime determines the current time and fills in the fields in the *timeb* structure pointed to by *buf*. The *timeb* structure contains four fields: *time, millitm, _timezone,* and *dstflag*:

```
struct timeb {
   long time ;
   short millitm ;
   short _timezone ;
   short dstflag ;
};
```

time	provides the time in seconds since 00:00:00 Greenwich mean time (GMT) January 1 1970.
millitm	is the fractional part of a second in milliseconds.
_timezone	is the difference in minutes between GMT and the local time. This value is computed going west from GMT. *ftime* gets this field from the global variable *_timezone* which is set by *tzset*.
dstflag	is set to nonzero if daylight saving time is taken into account during time calculations.

Note: *ftime* calls *tzset*. Therefore it isn't necessary to call *tzset* explicitly when you use *ftime*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_fullpath, _wfullpath

See also Example Portability

Syntax

```
#include <stdlib.h>
char * _fullpath(char *buffer, const char *path, int buflen);
wchar_t * _wfullpath(wchar_t *buffer, const wchar_t *path, int buflen);
```

Description

Converts a path name from relative to absolute.

_fullpath converts the relative path name in *path* to an absolute path name that is stored in the array of characters pointed to by *buffer*. The maximum number of characters that can be stored at *buffer* is *buflen*. The function returns NULL if the buffer isn't big enough to store the absolute path name or if the path contains an invalid drive letter.

If *buffer* is NULL, *_fullpath* allocates a buffer of up to _MAX_PATH characters. This buffer should be freed using *free* when it is no longer needed. _MAX_PATH is defined in stdlib.h.

Return Value

If successful the *_fullpath* function returns a pointer to the buffer containing the absolute path name.

On error, this function returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

fwrite

See also Example Portability

Syntax

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Writes to a stream.

fwrite appends *n* items of data each of length *size* bytes to the given output file. The data written begins at *ptr*. The total number of bytes written is *(n x size)*. *ptr* in the declarations is a pointer to any object.

Return Value

On successful completion *fwrite* returns the number of items (not bytes) actually written.

On error it returns a short count.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

gcvt

See also Example Portability

Syntax

```
#include <stdlib.h>
char *gcvt(double value, int ndec, char *buf);
```

Description

Converts floating-point number to a string.

gcvt converts *value* to a null-terminated ASCII string and stores the string in *buf*. It produces *ndec* significant digits in FORTRAN F format, if possible; otherwise, it returns the value in the *printf E* format (ready for printing). It might suppress trailing zeros.

Return Value

gcvt returns the address of the string pointed to by *buf*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

geninterrupt

See also Example Portability

Syntax

```
#include <dos.h>
void geninterrupt(int intr_num);
```

Description

Generates a software interrupt.

The *geninterrupt* macro triggers a software trap for the interrupt given by *intr_num.* The state of all registers after the call depends on the interrupt called.

Interrupts can leave registers in unpredictable states.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

getc, getwc

See also Example Portability

Syntax

```
#include <stdio.h>
int getc(FILE *stream);
wint_t getwc(FILE *stream);
```

Description

Gets character from stream.

getc returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Note: For Win32s or Win32 GUI applications, stdin must be redirected.

Return Value

On success, *getc* returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

getcbrk

See also Example Portability

Syntax

```
#include <dos.h>
int getcbrk(void);
```

Description

Gets control-break setting.

getcbrk uses the DOS system call 0x33 to return the current setting of control-break checking.

Return Value

getcbrk returns 0 if control-break checking is off, or 1 if checking is on.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

getch

See also Example Portability

Syntax

```
#include <conio.h>
int getch(void);
```

Description

Gets character from keyboard, does not echo to screen.

getch reads a single character directly from the keyboard, without echoing to the screen.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

getch returns the character read from the keyboard.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

getchar, getwchar

See also Example Portability

Syntax

```
#include <stdio.h>
int getchar(void);
wint_t getwchar(void);
```

Description

Gets character from stdin.

getchar is a macro that returns the next character on the named input stream stdin. It is defined to be *getc(stdin)*.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

On success, *getchar* returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

getche

See also Example Portability

Syntax

```
#include <conio.h>
int getche(void);
```

Description

Gets character from the keyboard, echoes to screen.

getche reads a single character from the keyboard and echoes it to the current text window using direct video or BIOS.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

getche returns the character read from the keyboard.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

getcurdir, _wgetcurdir

See also Example Portability

Syntax

```
#include <dir.h>
int getcurdir(int drive, char *directory);
int _wgetcurdir(int drive, wchar_t *directory );
```

Description

Gets current directory for specified drive.

getcurdir gets the name of the current working directory for the drive indicated by *drive*. *drive* specifies a drive number (0 for default, 1 for A, and so on). *directory* points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.

Return Value

getcurdir returns 0 on success or -1 in the event of error.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

getcwd, _wgetcwd

See also Example Portability

Syntax

```
#include <dir.h>
char *getcwd(char *buf, int buflen);
wchar_t *_wgetcwd(wchar_t *buf, int buflen);
```

Description

Gets current working directory.

getcwd gets the full path name (including the drive) of the current working directory, up to *buflen* bytes long and stores it in *buf*. If the full path name length (including the null terminator) is longer than *buflen* bytes, an error occurs.

If *buf* is NULL, a buffer *buflen* bytes long is allocated for you with *malloc*. You can later free the allocated buffer by passing the return value of *getcwd* to the function *free*.

Return Value

getcwd returns the following values:

- If *buf* is not NULL on input, *getcwd* returns *buf* on success, NULL on error.
- If *buf* is NULL on input, *getcwd* returns a pointer to the allocated buffer.

In the event of an error return, the global variable *errno* is set to one of the following values:

ENODEV	No such device
ENOMEM	Not enough memory to allocate a buffer (*buf* is NULL)
ERANGE	Directory name longer than *buflen* (*buf* is not NULL)

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_getdcwd, _wgetdcwd

See also Example Portability

Syntax

```
#include <direct.h>
char * _getdcwd(int drive, char *buffer, int buflen);
wchar_t * _wgetdcwd(int drive, wchar_t *buffer, int buflen);
```

Description

Gets current directory for specified drive.

_getdcwd gets the full path name of the working directory of the specified drive (including the drive name), up to *buflen* bytes long, and stores it in *buffer*. If the full path name length (including the null-terminator) is longer than *buflen*, an error occurs. The drive is 0 for the default drive, 1=A, 2=B, and so on.

If the working directory is the root directory, the terminating character for the full path is a backslash. If the working directory is a subdirectory, there is no terminating backslash after the subdirectory name.

If *buffer* is NULL, *_getdcwd* allocates a buffer at least *buflen* bytes long. You can later free the allocated buffer by passing the *_getdcwd* return value to the *free* function.

Return Value

If successful, *_getdcwd* returns a pointer to the buffer containing the current directory for the specified drive.

Otherwise it returns NULL, and sets the global variable errno to one of the following values:

ENOMEM	Not enough memory to allocate a buffer (*buffer* is NULL)
ERANGE	Directory name longer than *buflen* (*buffer* is not NULL)

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

getdfree

See also Example Portability

Syntax

```
#include <dos.h>
void getdfree(unsigned char drive, struct dfree *dtable);
```

Description

Gets disk free space.

getdfree accepts a drive specifier in *drive* (0 for default, 1 for A, and so on) and fills the *dfree* structure pointed to by *dtable* with disk attributes.

The *dfree* structure is defined as follows:

```
struct dfree {
   unsigned df_avail;     /* available clusters */
   unsigned df_total;     /* total clusters */
   unsigned df_bsec;      /* bytes per sector */
   unsigned df_sclus;     /* sectors per cluster */
};
```

Return Value

getdfree returns no value. In the event of an error, *df_sclus* in the *dfree* structure is set to (**unsigned**) -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

getdisk, setdisk

See also Examples Portability

Syntax

```
#include <dir.h>
int getdisk(void);
int setdisk(int drive);
```

Description

Gets or sets the current drive number.

getdisk gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.

setdisk sets the current drive to the one associated with drive: 0 for A, 1 for B, 2 for C, and so on.

The *setdisk* function changes the current drive of the parent process.

Return Value

getdisk returns the current drive number. *setdisk* returns the total number of drives available.

Examples

getdisk

setdisk

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

getdta

See also Example Portability

Syntax

```
#include <dos.h>
char far *getdta(void);
```

Description

Gets disk transfer address.

getdta returns the current setting of the disk transfer address (DTA).

In the small and medium memory models, the current data segment is the assumed segment. If you use C or C++ exclusively, this will be the case, but assembly routines can set the DTA to any hardware address.

In the compact or large models, the address returned by *getdta* is the correct hardware address and can be located outside the program.

Return Value

getdta returns a far pointer to the current DTA.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

getenv, _wgetenv

See also Example Portability

Syntax

```
#include <stdlib.h>
char *getenv(const char *name);
wchar_t *_wgetenv(const wchar_t *name);
```

Description

Find or delete an environment variable from the system environment.

The environment consists of a series of entries that are of the form `name=string\0`.

getenv returns the value of a specified variable. On DOS and OS/2, *name* must be uppercase. On other systems, *name* can be either uppercase or lowercase. *name* must not include the equal sign (**=**). If the specified environment variable does not exist, *getenv* returns a NULL pointer.

To delete the variable from the environment, use `getenv("name=")`.

Note: Environment entries must not be changed directly. If you want to change an environment value, you must use *putenv*.

Return Value

On success, *getenv* returns the value associated with *name*.

If the specified *name* is not defined in the environment, *getenv* returns a NULL pointer.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

getfat

See also Example Portability

Syntax

```
#include <dos.h>
void getfat(unsigned char drive, struct fatinfo *dtable);
```

Description

Gets file allocation table information for given drive.

getfat gets information from the file allocation table (FAT) for the drive specified by *drive* (0 for default, 1 for A, 2 for B, and so on). *dtable* points to the *fatinfo* structure to be filled in. The *fatinfo* structure filled in by *getfat* is defined as follows:

```
struct fatinfo {
   char fi_sclus;          /* sectors per cluster */
   char fi_fatid;          /* the FAT id byte */
   unsigned fi_nclus;      /* number of clusters */
   int fi_bysec;           /* bytes per sector */
};
```

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

getfatd

See also Example Portability

Syntax

```
#include <dos.h>
void getfatd(struct fatinfo *dtable);
```

Description

Gets file allocation table information.

getfatd gets information from the file allocation table (FAT) of the default drive. *dtable* points to the *fatinfo* structure to be filled in.

The *fatinfo* structure filled in by *getfatd* is defined as follows:

```
struct fatinfo {
   char fi_sclus;     /* sectors per cluster */
   char fi_fatid;     /* the FAT id byte */
   int fi_nclus;      /* number of clusters */
   int fi_bysec;      /* bytes per sector */
};
```

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

getftime, setftime

See also Examples Portability

Syntax

```
#include <io.h>
int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);
```

Description

Gets and sets the file date and time.

getftime retrieves the file time and date for the disk file associated with the open *handle*. The *ftime* structure pointed to by *ftimep* is filled in with the file's time and date.

setftime sets the file date and time of the disk file associated with the open *handle* to the date and time in the *ftime* structure pointed to by *ftimep*. The file must not be written to after the *setftime* call or the changed information will be lost. The file must be open for writing; an EACCES error will occur if the file is open for read-only access.

setftime requires the file to be open for writing; an EACCES error will occur if the file is open for read-only access.

The *ftime* structure is defined as follows:

```
struct ftime {
   unsigned ft_tsec: 5;      /* two seconds */
   unsigned ft_min: 6;       /* minutes */
   unsigned ft_hour: 5;      /* hours */
   unsigned ft_day: 5;       /* days */
   unsigned ft_month: 4;     /* months */
   unsigned ft_year: 7;      /* year - 1980*/
   };
```

Return Value

getftime and *setftime* return 0 on success.

In the event of an error return -1 is returned and the global variable *errno* is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
EINVFNC	Invalid function number

Examples

getftime

setftime

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_get_osfhandle

See also Example Portability

Syntax

```
#include <io.h>
long _get_osfhandle(int filehandle);
```

Description

Associates file handles.

The _get_osfhandle function associates an operating system file handle with an existing run-time file handle. The variable *filehandle* is the file handle of your program.

Return value

On success, _get_osfhandle returns an operating system file handle corresponding to the variable *filehandle*.

On error, it returns -1 and sets the global variable errno to

EBADF	an invalid file handle

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			

getpass

See also Example Portability

Syntax

```
#include <conio.h>
char *getpass(const char *prompt);
```

Description

Reads a password.

getpass reads a password from the system console after prompting with the null-terminated string *prompt* and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null-terminator).

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

The return value is a pointer to a static string which is overwritten with each call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+			+

getpid

See also Example Portability

Syntax

```
#include <process.h>
unsigned getpid(void)
```

Description

Gets the process ID of a program.

This function returns the current process ID--an integer that uniquely identifies the process.

Return Value

getpid returns the current process' ID.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

getpsp

See also Example Portability

Syntax

```
#include <dos.h>
unsigned getpsp(void);
```

Description

Gets the program segment prefix.

getpsp gets the segment address of the program segment prefix (PSP) using DOS call 0x62.

Return Value

getpsp returns the address of the Program Segment Prefix (PSP).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

gets, _getws

See also Example Portability

Syntax

```
#include <stdio.h>
char *gets(char *s);
wchar_t *_getws(wchar_t *s);  // Unicode version
```

Description

Gets a string from stdin.

gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into *s*. The new line is replaced by a null character (\0) in *s*.

gets allows input strings to contain certain whitespace characters (spaces, tabs). *gets* returns when it encounters a new line; everything up to the new line is copied into *s*.

The *gets* function is not length-terminated. If the input string is sufficiently large, data can be overwritten and corrupted. The *fgets* function provides better control of input strings.

Note: For Win32s or Win32 GUI applications, stdin must be redirected.

Return Value

On success, *gets* returns the string argument *s*.

On end-of-file or error, it returns NULL

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

gettext

See also Example Portability

Syntax

```
#include <conio.h>
int gettext(int left, int top, int right, int bottom, void *destin);
```

Description

Copies text from text mode screen to memory.

gettext stores the contents of an onscreen text rectangle defined by *left*, *top*, *right*, and *bottom* into the area of memory pointed to by *destin*.

All coordinates are absolute screen coordinates not window-relative. The upper left corner is (1,1). *gettext* reads the contents of the rectangle into memory sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell and the second is the cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as

bytes = *(h* rows) x *(w* columns) x 2

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

gettext returns 1 if the operation succeeds.

On error, it returns 0 (for example, if it fails because you gave coordinates outside the range of the current screen mode).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

gettextinfo

See also Example Portability

Syntax

```
#include <conio.h>
void gettextinfo(struct text_info *r);
```

Description

Gets text mode video information.

gettextinfo fills in the *text_info* structure pointed to by *r* with the current text video information.

The *text_info* structure is defined in conio.h as follows:

```
struct text_info {
   unsigned char winleft;        /* left window coordinate */
   unsigned char wintop;         /* top window coordinate */
   unsigned char winright;       /* right window coordinate */
   unsigned char winbottom;      /* bottom window coordinate */
   unsigned char attribute;      /* text attribute */
   unsigned char normattr;       /* normal attribute */
   unsigned char currmode;       /* BW40, BW80, C40, C80, or C4350 */
   unsigned char screenheight;   /* text screen's height */
   unsigned char screenwidth;    /* text screen's width */
   unsigned char curx;           /* x-coordinate in current window */
   unsigned char cury;           /* y-coordinate in current window */
};
```

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None. Results are returned in the structure pointed to by *r*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

gettime, settime

See also Examples Portability

Syntax

```
#include <dos.h>
void gettime(struct time *timep);
void settime(struct time *timep);
```

Description

Gets and sets the system time.

gettime fills in the *time* structure pointed to by *timep* with the system's current time.

settime sets the system time to the values in the *time* structure pointed to by *timep*.

The *time* structure is defined as follows:

```
struct time {
   unsigned char ti_min;      /* minutes */
   unsigned char ti_hour;     /* hours */
   unsigned char ti_hund;     /* hundredths of seconds */
   unsigned char ti_sec;      /* seconds */
};
```

Return Value

None.

Examples

gettime

settime

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
gettime	+		+	+			+
settime	+		+				+

getvect, setvect

See also Example Portability

Syntax

```
#include <dos.h>
void interrupt(*getvect(int interruptno)) ();                  /* C version
*/
void interrupt(*getvect(int interruptno)) ( ... );             // C++ version
void setvect(int interruptno, void interrupt (*isr) ());       /* C version
*/
void setvect(int interruptno, void interrupt (*isr) ( ... ));  // C++ version
```

Description

Gets and sets interrupt vector.

Every processor of the 8086 family includes a set of interrupt vectors numbered 0 to 255. The 4-byte value in each vector is actually an address which is the location of an interrupt function.

getvect reads the value of the interrupt vector given by *interruptno* and returns that value as a (far) pointer to an interrupt function. The value of *interruptno* can be from 0 to 255.

setvect sets the value of the interrupt vector named by *interruptno* to a new value, *isr*, which is a far pointer containing the address of a new interrupt function. The address of a C routine can be passed to *isr* only if that routine is declared to be an interrupt routine.

Note: In C++, only static member functions or non-member functions can be declared to be an interrupt routine. If you use the prototypes declared in dos.h, simply pass the address of an interrupt function to *setvect* in any memory model

Return Value

getvect returns the current 4-byte value stored in the interrupt vector named by *interruptno*.

set*v*ect does not return a value.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

getverify

See also Example Portability

Syntax

```
#include <dos.h>
int getverify(void);
```

Description

Returns the state of the operating system verify flag.

getverify gets the current state of the verify flag.

The verify flag controls output to the disk. When verify is off writes are not verified; when verify is on all disk writes are verified to ensure proper writing of the data.

Return Value

getverify returns the current state of the verify flag either 0 (off) or 1 (on).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

_getw

See also Example Portability

Syntax

```
#include <stdio.h>
int _getw(FILE *stream);
```

Description

Gets an integer from stream.

_getw returns the next integer in the named input stream. It assumes no special alignment in the file.

_getw should not be used when the stream is opened in text mode.

Return Value

_getw returns the next integer on the input stream.

On end-of-file or error, *_getw* returns EOF.

Note: Because EOF is a legitimate value for *_getw* to return, feof or ferror should be used to detect end-of-file or error.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

gmtime

See also Example Portability

Syntax

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Description

Converts date and time to Greenwich mean time (GMT).

gmtime accepts the address of a value returned by *time* and returns a pointer to the structure of type *tm* containing the time elements. *gmtime* converts directly to GMT.

The global long variable *_timezone* should be set to the difference in seconds between GMT and local standard time (in PST *_timezone* is 8 x 60 x 60). The global variable _daylight should be set to nonzero *only* if the standard U.S. daylight saving time conversion should be applied.

This is the *tm* structure declaration from the time.h header file:

```
struct tm {
   int tm_sec;             /* Seconds */
   int tm_min;             /* Minutes */
   int tm_hour;            /* Hour (0 - 23) */
   int tm_mday;            /* Day of month (1 - 31) */
   int tm_mon;             /* Month (0 - 11) */
   int tm_year;            /* Year (calendar year minus 1900) */
   int tm_wday;            /* Weekday (0 - 6; Sunday is 0) */
   int tm_yday;            /* Day of year (0 -365) */
   int tm_isdst;           /* Nonzero if daylight saving time is in effect.
  */
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if daylight saving time is in effect.

Return Value

gmtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

gotoxy

See also Example Portability

Syntax

```
#include <conio.h>
void gotoxy(int x
int y);
```

Description

Positions cursor in text window.

gotoxy moves the cursor to the given position in the current text window. If the coordinates are in any way invalid the call to *gotoxy* is ignored. An example of this is a call to *gotoxy*(40,30) when (35,25) is the bottom right position in the window. Neither argument to *gotoxy* can be zero.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Examples

gotoxygotoxy_Ex

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

heapcheck

Example Portability

Syntax

```
#include <alloc.h>
int heapcheck(void);
```

Description

Checks and verifies the heap.

heapcheck walks through the heap and examines each block, checking its pointers, size, and other critical attributes. For DOS users, *heapcheck* maps to *farheapcheck* in the large and huge memory models.

Return Value

The return value is less than 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is verified

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

heapcheckfree

Example Portability

Syntax

```
#include <alloc.h>
int heapcheckfree(unsigned int fillvalue);
```

Description

Checks the free blocks on the heap for a constant value.

Return Value

The return value is less then 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_BADVALUE	A value other than the fill value was found
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

heapchecknode

Example Portability

Syntax

```
#include <alloc.h>
int heapchecknode(void *node);
```

Description

Checks and verifies a single node on the heap.

If a node has been freed and *heapchecknode* is called with a pointer to the freed block, *heapchecknode* can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists.

The *heapchecknode* function should be used only to inspect allocations which were created by *malloc* or *calloc*.

Return Value

One of the following values:

_BADNODE	Node could not be found
_FREEENTRY	Node is a free block
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_USEDENTRY	Node is a used block

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_heapchk

See also Example Portability

Syntax

```
#include <malloc.h>
int _heapchk(void);
```

Description

Checks and verifies the heap.

_heapchk walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

Return Value

One of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPEMPTY	No heap exists
_HEAPOK	The heap appears to be uncorrupted

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

heapfillfree

Example Portability

Syntax

```
#include <alloc.h>
int heapfillfree(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

Return Value

One of the following values:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_heapmin

See also Portability

Syntax

```
#include <malloc.h>
int _heapmin(void);
```

Description

Release unused heap areas.

The *_heapmin* function returns unused areas of the heap to the operating system. This allows blocks that have been allocated and then freed to be used by other processes. Due to fragmentation of the heap, *_heapmin* might not always be able to return unused memory to the operating system; this is not an error.

Return Value

_heapmin returns 0 if it is successful, or -1 if an error occurs.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_heapset

See also Example Portability

Syntax

```
#include <malloc.h>
int _heapset(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

_heapset checks the heap for consistency using the same methods as _heapchk. It then fills each free block in the heap with the value contained in the least significant byte of *fillvalue*. This function can be used to find heap-related problems. It does *not* guarantee that subsequently allocated blocks will be filled with the specified value.

Return Value

One of the following values:

_HEAPOK	The heap appears to be uncorrupted
_HEAPEMPTY	No heap exists
_HEAPBADNODE	A corrupted heap block has been found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

heapwalk

Example Portability

Syntax

```
#include <alloc.h>
int heapwalk(struct heapinfo *hi);
```

Description

heapwalk is used to "walk" through the heap, node by node.

heapwalk assumes the heap is correct. Use heapcheck to verify the heap before using *heapwalk*. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to *heapwalk*.

heapwalk receives a pointer to a structure of type *heapinfo* (declared in alloc.h). For the first call to *heapwalk*, set the hi.ptr field to null. *heapwalk* returns with hi.ptr containing the address of the first block. hi.size holds the size of the block in bytes. hi.in_use is a flag that's set if the block is currently in use.

Return Value

One of the following values:

_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The *heapinfo* block contains valid information about the next heap block

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

highvideo

See also Example Portability

Syntax

```
#include <conio.h>
void highvideo(void);
```

Description

Selects high-intensity characters.

highvideo selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen, but does affect those displayed by functions (such as *cprintf*) that perform direct video, text mode output after *highvideo* is called.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

hypot, hypotl

Example Portability

Syntax

```
#include <math.h>
double hypot(double x, double y);
long double hypotl(long double x, long double y);
```

Description

Calculates hypotenuse of a right triangle.

hypot calculates the value *z* where

$z^2 = x^2 + y^2$ and $z >= 0$

This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are *x* and *y*.

hypotl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, these functions return *z*, a **double** (*hypot*) or a **long double**) (*hypotl*). On error (such as an overflow), they set the global variable errno to

ERANGE	Result out of range

and return the value HUGE_VAL (*hypot*) or _LHUGE_VAL) (*hypotl*). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
hypot	+	+	+	+			+
hypotl	+		+	+			+

_InitEasyWin

See also Example Portability

Syntax

```
#include <io.h>
void _InitEasyWin(void);
```

Description

The purpose of EasyWin is to convert DOS applications to Windows programs, quickly and easily. You might, however, occasionally want to use EasyWin from within a true 16-bit Windows program. For example, you might want to add printf functions to your program code to help you debug your 16-bit Windows program.

To use EasyWin from within a Windows program, make a call to the *_InitEasyWin* function before doing any standard input or output.

For example:

```
#include <windows.h>
#include <stdio.h>

#pragma argsused
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpszCmdLine, int cmdShow)
{
   _InitEasyWin();

   /* Normal windows setup */

   printf("Hello, world\n");
   return 0;
}
```

The prototype for *_InitEasyWin* can be found in stdio.h and iostream.h.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
		+				

inp

See also Example Portability

Syntax

```
#include <conio.h>
int inp(unsigned portid);
```

Description

Reads a byte from a hardware port.

inp is a macro that reads a byte from the input port specified by *portid*. If *inp* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and undefine the macro *inp*, you get the *inp* function.

Return Value

inp returns the value read.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

inport

See also Example Portability

Syntax

```
#include <dos.h>
int inport(int portid);
```

Description

Reads a word from a hardware port.

inport works just like the 80x86 instruction *IN*. It reads the low byte of a word from the input port specified by *portid*; it reads the high byte from *portid* + 1.

Return Value

inport returns the value read.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

inportb

See also Example Portability

Syntax

```
#include <dos.h>
unsigned char inportb(int portid);
```

Description

Reads a byte from a hardware port.

inportb is a macro that reads a byte from the input port specified by *portid*.

If *inportb* is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include dos.h and **#undef** the macro *inportb*, you get the *inportb* function.

Return Value

inportb returns the value read.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

inpw

See also Example Portability

Syntax

```
#include <conio.h>
unsigned inpw(unsigned portid);
```

Description

Reads a word from a hardware port.

inpw is a macro that reads a 16-bit word from the inport port specified by *portid*. It reads the low byte of the word from *portid*, and the high byte from *portid* + 1.

If *inpw* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and **#undef** the macro *inpw*, you get the *inpw* function.

Return Value

inpw returns the value read.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

insline

See also Example Portability

Syntax

```
#include <conio.h>
void insline(void);
```

Description

Inserts a blank line in the text window.

insline inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line, and the bottom line scrolls off the bottom of the window.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

int86

See also Example Portability

Syntax

```
#include <dos.h>
int int86(int intno, union REGS *inregs, union REGS *outregs);
```

Description

General 8086 software interrupt.

int86 executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

After the software interrupt returns, *int86* copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it usually indicates that an error has occurred.

Note: *inregs* can point to the same structure that *outregs* points to.

Return Value

int86 returns the value of AX after completion of the software interrupt. If the carry flag is set (outregs -> x.cflag != 0), indicating an error, this function sets the global variable _doserrno to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

int86x

See also Example Portability

Syntax

```
#include <dos.h>
int int86x(int intno, union REGS *inregs, union REGS *outregs, struct SREGS
  *segregs);
```

Description

General 8086 software interrupt interface.

int86x executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

In addition, *int86x* copies the *segregs ->ds* and *segregs ->es* values into the corresponding registers before executing the software interrupt. This feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the software interrupt.

After the software interrupt returns, *int86x* copies the current register values to *outregs*, the status of the carry flag to the x.cflag field in *outregs*, and the value of the 8086 flags register to the x.flags field in *outregs*. In addition, *int86x* restores DS and sets the *segregs ->es* and *segregs ->ds* fields to the values of the corresponding segment registers. If the carry flag is set, it usually indicates that an error has occurred.

int86x lets you invoke an 8086 software interrupt that takes a value of DS different from the default data segment, and/or takes an argument in ES.

Note: *inregs* can point to the same structure that *outregs* points to.

Return Value

int86x returns the value of AX after completion of the software interrupt. If the carry flag is set (outregs -> x.cflag != 0), indicating an error, this function sets the global variable _doserrno to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

intdos

See also Example Portability

Syntax

```
#include <dos.h>
int intdos(union REGS *inregs, union REGS *outregs);
```

Description

General DOS interrupt interface.

intdos executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.ah* specifies the DOS function to be invoked.

After the interrupt 0x21 returns, *intdos* copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it indicates that an error has occurred.

Note: *inregs* can point to the same structure that outregs points to.

Return Value

intdos returns the value of AX after completion of the DOS function call. If the carry flag is set (outregs -> x.cflag != 0), indicating an error, it sets the global variable _doserrno to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

intdosx

See also Example Portability

Syntax

```
#include <dos.h>
int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
```

Description

General DOS interrupt interface.

intdosx executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.ah* specifies the DOS function to be invoked.

In addition, *intdosx* copies the *segregs ->ds* and *segregs ->es* values into the corresponding registers before invoking the DOS function. This feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the function execution.

After the interrupt 0x21 returns, *intdosx* copies the current register values to *outregs*, copies the status of the carry flag to the x.cflag field in *outregs*, and copies the value of the 8086 flags register to the x.flags field in *outregs*. In addition, *intdosx* sets the *segregs ->es* and *segregs ->ds* fields to the values of the corresponding segment registers and then restores DS. If the carry flag is set, it indicates that an error occurred.

intdosx lets you invoke a DOS function that takes a value of DS different from the default data segment and/or takes an argument in ES.

Note: *inregs* can point to the same structure that *outregs* points to.

Return Value

intdosx returns the value of AX after completion of the DOS function call. If the carry flag is set (outregs -> x.cflag != 0), indicating an error, it sets the global variable _doserrno to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

intr

See also Example Portability

Syntax

```
#include <dos.h>
void intr(int intno, struct REGPACK *preg);
```

Description

Alternate 8086 software interrupt interface.

The *intr* function is an alternate interface for executing software interrupts. It generates an 8086 software interrupt specified by the argument *intno*.

intr copies register values from the REGPACK structure **preg* into the registers before executing the software interrupt. After the software interrupt completes, *intr* copies the current register values into **preg*, including the flags.

The arguments passed to *intr* are as follows:

intno	Interrupt number to be executed
preg	Address of a structure containing
	(a) the input registers before the interrupt call
	(b) the value of the registers after the interrupt call

The REGPACK structure (defined in dos.h) has the following format:

```
struct  REGPACK {
   unsigned  r_ax, r_bx, r_cx, r_dx;
   unsigned  r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

Return Value

No value is returned. The REGPACK structure **preg* contains the value of the registers after the interrupt call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

ioctl

Example Portability

Syntax

```
#include <io.h>
int ioctl(int handle, int func [, void *argdx, int argcx]);
```

Description

Controls I/0 device.

ioctl is available on UNIX systems, but not with these parameters or functionality. UNIX version 7 and System III differ from each other in their use of *ioctl*. *ioctl* calls are not portable to UNIX and are rarely portable across DOS machines.

DOS 3.0 extends *ioctl* with *func* values of 8 and 11.

This is a direct interface to the DOS call 0x44 (IOCTL).

The exact function depends on the value of *func* as follows:

0	Get device information.
1	Set device information (in *argdx*).
2	Read *argcx* bytes into the address pointed to by *argdx*.
3	Write *argcx* bytes from the address pointed to by *argdx*.
4	Same as 2 except *handle* is treated as a drive number (0 equals default, 1 equals A, and so on).
5	Same as 3 except *handle* is a drive number (0 equals default, 1 equals A, and so on).
6	Get input status.
7	Get output status.
8	Test removability; DOS 3.0 only.
11	Set sharing conflict retry count; DOS 3.0 only.

ioctl can be used to get information about device channels. Regular files can also be used, but only *func* values 0, 6, and 7 are defined for them. All other calls return an EINVAL error for files.

See the documentation for system call 0x44 in your DOS reference manuals for detailed information on argument or return values.

The arguments *argdx* and *argcx* are optional.

ioctl provides a direct interface to DOS device drivers for special functions. As a result, the exact behavior of this function varies across different vendors' hardware and in different devices. Also, several vendors do not follow the interfaces described here. Refer to the vendor BIOS documentation for exact use of *ioctl*.

Return Value

For *func* 0 or 1, the return value is the device information (DX of the *ioctl* call). For *func* values of 2 through 5, the return value is the number of bytes actually transferred. For *func* values of 6 or 7, the return value is the device status.

In any event, if an error is detected, a value of -1 is returned, and the global variable errno is set to one of the following:

EBADF	Bad file number
EINVAL	Invalid argument
EINVDAT	Invalid data

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+				

isalnum, iswalnum, _ismbcalnum

Example Portability

Syntax

```
#include <ctype.h>
int isalnum(int c);
int iswalnum(wint_t c);

#include <mbstring.h>
int _ismbcalnum(unsigned int c);
```

Description

Tests for an alphanumeric character.

isalnum is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a letter (*A* to *Z* or *a* to *z*) or a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

It is a predicate returning nonzero for true and 0 for false. *isalnum* returns nonzero if *c* is a letter or a digit.

iswalnum returns nonzero if *iswalpha* or *iswdigit* return true for *c*.

_ismbcalnum returns true if and only if the argument *c* is a single-byte ASCII English letter.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isalpha, iswalpha, _ismbcalpha

Example Portability

Syntax

```
#include <ctype.h>
int isalpha(int c);
int iswalpha(wint_t c);

#include <mbstring.h>
int _ismbcalpha(unsigned int c);
```

Description

Classifies an alphabetical character.

isalpha is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a letter (*A* to *Z* or *a* to *z*).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isalpha returns nonzero if *c* is a letter.

iswalpha returns nonzero if *c* is a **wchar_t** in the character set defined by the implementation.

_ismbcalpha returns true if and only if the argument *c* is a single-byte ASCII English letter.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isascii, iswascii

Example Portability

Syntax

```
#include <ctype.h>
int isascii(int c);
int iswascii(wint_t c);
```

Description

Character classification macro.

These functions depend on the LC_CTYPE

isascii is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.

isascii is defined on all integer values.

Return Value

isascii returns nonzero if *c* is in the range 0 to 127 (0x00-0x7F).

iswascii returns nonzero if *c* is

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

isatty

Example Portability

Syntax

```
#include <io.h>
int isatty(int handle);
```

Description

Checks for device type.

isatty determines whether *handle* is associated with any one of the following character devices:

- a terminal
- a console
- a printer
- a serial port

Return Value

If the device is one of the four character devices listed above, *isatty* returns a nonzero integer. If it is not such a device, *isatty* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

iscntrl, iswcntrl

Example Portability

Syntax

```
#include <ctype.h>
int iscntrl(int c);
int iswcntrl(wint_t c);
```

Description

Tests for a control character.

iscntrl is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a delete character or control character (0x7F or 0x00 to 0x1F).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

iscntrl returns nonzero if *c* is a delete character or ordinary control character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isdigit, iswdigit, _ismbcdigit

Example Portability

Syntax

```
#include <ctype.h>
int isdigit(int c);
int iswdigit(wint_t c);

#include <mbstring.h>
int _ismbcdigit(unsigned int c);
```

Description

Tests for decimal-digit character.

isdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isdigit returns nonzero if *c* is a digit.

_ismbcdigit returns true if and only if the argument *c* is a single-byte representation of an ASCII digit.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isgraph, iswgraph, _ismbcgraph

Example Portability

Syntax

```
#include <ctype.h>
int isgraph(int c);
int iswgraph(wint_t c);

#include <mbstring.h>
int _ismbcgraph( unsigned int c);
```

Description

Tests for printing character.

isgraph is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a printing character except blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isgraph returns nonzero if *c* is a printing character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

islower, iswlower, _ismbclower

Example Portability

Syntax

```
#include <ctype.h>
int islower(int c);
int iswlower(wint_t c);

#include <mbstring.h>
int _ismbclower(unsigned int c);
```

Description

Tests for lowercase character.

islower is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a lowercase letter (*a* to *z*).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

islower returns nonzero if *c* is a lowercase letter.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_ismbblead, _ismbbtrail

Syntax

```
#include <mbstring.h>
int _ismbblead(unsigned int c);
int _ismbbtrail(unsigned int c);
```

Description

_ismbblead and *_ismbbtrail* are used to test whether the argument *c* is the first or the second byte of a multibyte character.

_ismbblead and *_ismbbtrail* are affected by the code page in use. You can set the code page by using the *_setlocale* function.

Return Value

If *c* is in the lead byte of a multibyte character, *_ismbblead* returns true.

If *c* is in the trail byte of a multibyte character, *_ismbbtrail* returns a nonzero value.

_ismbclegal

Syntax

```
#include <mbstring.h>
int _ismbclegal(unsigned int c);
```

Description

_ismbclegal tests whether each byte of the *c* argument is in the code page that is currently in use.

Return Value

_ismbclegal returns a nonzero value if the argument *c* is a valid multibyte character on the current code page. Otherwise, the function returns zero.

_ismbslead, _ismbstrail

See also

Syntax

```
#include <mbstring.h>
int _ismbslead(const unsigned char *s1, const unsigned char *s2);
int _ismbstrail(const unsigned char *s1, const unsigned char *s2);
```

Description

The *_ismbslead* and *_ismbstrail* functions test the *s1* argument to determine whether the *s2* argument is a pointer to the lead byte or the trail byte. The test is case-sensitive.

Return Value

The *_ismbslead* and *_ismbstrail* routines return -1 if *s2* points to a lead byte or a trail byte, respectively. If the test is false, the routines return zero.

isprint, iswprint, _ismbcprint

Example Portability

Syntax

```
#include <ctype.h>
int isprint(int c);
int iswprint(wint_t c);

#include <mbstring.h>
int _ismbcprint(unsigned int c);
```

Description

Tests for printing character.

isprint is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a printing character including the blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isprint returns nonzero if *c* is a printing character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ispunct, iswpunct, _ismbcpunct

Example Portability

Syntax

```
#include <ctype.h>
int ispunct(int c);
int iswpunct(wint_t c);

#include <mbstring.h>
int _ismbcpunct(unsigned int c);
```

Description

Tests for punctuation character.

ispunct is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is any printing character that is neither an alphanumeric nor a blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

ispunct returns nonzero if *c* is a punctuation character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isspace, iswspace, _ismbcspace

Example Portability

Syntax

```
#include <ctype.h>
int isspace(int c);
int iswspace(wint_t c);

#include <mbstring.h>
int _ismbcspace(unsigned int c);
```

Description

Tests for space character.

isspace is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isspace returns nonzero if *c* is a space, tab, carriage return, new line, vertical tab, formfeed (0x09 to 0x0D, 0x20), or any other locale-defined space character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isupper, iswupper, _ismbcupper

Example Portability

Syntax

```
#include <ctype.h>
int isupper(int c);
int iswupper(wint_t c);

#include <mbstring.h>
int _ismbcupper(unsigned int c);
```

Description

Tests for uppercase character.

isupper is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is an uppercase letter (*A* to *Z*).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isupper returns nonzero if *c* is an uppercase letter.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isxdigit, iswxdigit

Example Portability

Syntax

```
#include <ctype.h>
int isxdigit(int c);
int iswxdigit(wint_t c);
```

Description

Tests for hexadecimal character.

isxdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (*#undef*) it.

Return Value

isxdigit returns nonzero if *c* is a hexadecimal digit (0 to 9, *A* to *F*, *a* to *f*) or any other hexadecimal digit defined by the locale.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

itoa, _itow

See also Example Portability

Syntax

```
#include <stdlib.h>
char *itoa(int value, char *string, int radix);
wchar_t *_itow(int value, wchar_t *string, int radix);
```

Description

Converts an integer to a string.

itoa converts *value* to a null-terminated string and stores the result in *string*. With *itoa*, *value* is an integer.

radix specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (-).

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). *itoa* can return up to 17 bytes.

Return Value

itoa returns a pointer to *string*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

kbhit

See also Example Portability

Syntax

```
#include <conio.h>
int kbhit(void);
```

Description

Checks for currently available keystrokes.

kbhit checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with *getch* or *getche*.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

If a keystroke is available, *kbhit* returns a nonzero value. Otherwise, it returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

labs

See also Example Portability

Syntax

```
#include <math.h>
long labs(long int x);
```

Description

Gives long absolute value.

labs computes the absolute value of the parameter *x*.

Return Value

labs returns the absolute value of *x*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ldexp, ldexpl

See also Example Portability

Syntax

```
#include <math.h>
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
```

Description

Calculates $x * 2^{exp}$

lexpl is the **long double** version; it takes a **long double** argument for *x* and returns a **long double** result.

Return Value

On success, *ldexp* (or *ldexpl*) returns the value it calculated, $x * 2^{exp}$. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
ldexp	+	+	+	+	+	+	+
ldexpl	+		+	+			+

ldiv

See also Example Portability

Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

Divides two **longs**, returning quotient and remainder.

ldiv divides two **longs** and returns both the quotient and the remainder as an *ldiv_t* type. *numer* and *denom* are the numerator and denominator, respectively.

The *ldiv_t* type is a structure of **long**s defined in stdlib.h as follows:

```
typedef struct {
   long int quot;      /* quotient */
   long int rem;       /* remainder */
   } ldiv_t;
```

Return Value

ldiv returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

lfind

See also Example Portability

Syntax

```
#include <stdlib.h>
void  *lfind(const void *key, const void *base, size_t *num, size_t width,
  int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lfind makes a linear search for the value of *key* in an array of sequential records. It uses a user-defined comparison routine *fcmp*. The *fcmp* function must be used with the _USERENTRY calling convention.

The array is described as having **num* records that are *width* bytes wide, and begins at the memory location pointed to by *base*.

Return Value

lfind returns the address of the first entry in the table that matches the search key. If no match is found, *lfind* returns NULL. The comparison routine must return 0 if **elem1* == **elem2*, and nonzero otherwise (*elem1* and *elem2* are its two parameters).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

localeconv

See also Example Portability

Syntax

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

Queries the locale for numeric format.

This function provides information about the monetary and other numeric formats for the current locale. The information is stored in a **struct** *lconv* type. The structure can only be modified by the *setlocale*. Subsequent calls to *localeconv* will update the *lconv* structure.

The *lconv* structure is defined in locale.h. It contains the following fields:

Field	Application
char **decimal_point;*	Decimal point used in nonmonetary formats. This can never be an empty string.
char **thousands_sep;*	Separator used to group digits to the left of the decimal point. Not used with monetary quantities.
char **grouping;*	Size of each group of digits. Not used with monetary quantities. See the value listing table below.
char **int_curr_symbol;*	International monetary symbol in the current locale. The symbol format is specified in the ISO 4217 Codes for the Representation of Currency and Funds.
char **currency_symbol;*	Local monetary symbol for the current locale.
char **mon_decimal_point;*	Decimal point used to format monetary quantities.
char **mon_thousands_sep;*	Separator used to group digits to the left of the decimal point for monetary quantities.
char **mon_grouping;*	Size of each group of digits used in monetary quantities. See the value listing table below.
char **positive_sign;*	String indicating nonnegative monetary quantities.
char **negative_sign;*	String indicating negative monetary quantities.
char *int_frac_digits;*	Number of digits after the decimal point that are to be displayed in an internationally formatted monetary quantity.
char *frac_digits;*	Number of digits after the decimal point that are to be displayed in a formatted monetary quantity.
char *p_cs_precedes;*	Set to 1 if *currency_symbol* precedes a nonnegative formatted monetary quantity. If *currency_symbol* is after the quantity, it is set to 0.
char *p_sep_by_space;*	Set to 1 if *currency_symbol* is to be separated from the nonnegative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char *n_cs_precedes;*	Set to 1 if *currency_symbol* precedes a negative formatted monetary quantity. If *currency_symbol* is after the quantity, set to 0.
char *n_sep_by_space;*	Set to 1 if *currency_symbol* is to be separated from the negative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char *p_sign_posn;*	Indicate where to position the positive sign in a nonnegative formatted monetary quantity.

char *n_sign_posn;*	Indicate where to position the positive sign in a negative formatted monetary quantity.

Any of the above strings (except *decimal_point*) that is empty " " is not supported in the current locale. The nonstring **char** elements are nonnegative numbers. Any nonstring **char** element that is set to CHAR_MAX indicates that the element is not supported in the current locale.

The *grouping* and *mon_grouping* elements are set and interpreted as follows:

Value	Meaning
CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be used repeatedly for the remainder of the digits.
any other integer	Indicates how many digits make up the current group. The next element is read to determine the size of the next group of digits before the current group.

The *p_sign_posn* and *n_sign_posn* elements are set and interpreted as follows:

Value	Meaning
0	Use parantheses to surround the quantity and currency_symbol.
1	Sign string precedes the quantity and currency_symbol.
2	Sign string succeeds the quatity and currency_symbol.
3	Sign string immediately precedes the quantity and currency_symbol.
4	Sign string immediately succeeds the quantity and currency_symbol.

Return Value

Returns a pointer to the the filled-in structure of type **struct** *lconv*. The values in the structure will change whenever *setlocale* modifies the LC_MONETARY or LC_NUMERIC categories.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

localtime

See also Example Portability

Syntax

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Description

Converts date and time to a structure.

localtime accepts the address of a value returned by *time* and returns a pointer to the structure of type *tm* containing the time elements. It corrects for the time zone and possible daylight saving time.

The global long variable *_timezone* contains the difference in seconds between GMT and local standard time (in PST, *_timezone* is 8 x 60 x 60). The global variable *daylight* contains nonzero *only if* the standard U.S. daylight saving time conversion should be applied. These values are set by *tzset*, not by the user program directly.

This is the **tm** structure declaration from the time.h header file:

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if _daylight saving time is in effect.

Return Value

localtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

lock

See also Example Portability

Syntax

```
#include <io.h>
int lock(int handle, long offset, long length);
```

Description

Sets file-sharing locks. DOS users must be sure to load SHARE.EXE before using *lock.*

lock provides an interface to the operating system file-sharing mechanism.

A lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

Return Value

lock returns 0 on success. On error, *lock* returns -1 and sets the global variable errno to

EACCES	Locking violation

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

locking

See also Example Portability

Syntax

```
#include <io.h>
#include <sys\locking.h>
int locking(int handle, int cmd, long length);
```

Description

Sets or resets file-sharing locks. DOS users must be sure to load SHARE.EXE before using *locking.*

locking provides an interface to the operating system file-sharing mechanism. The file to be locked or unlocked is the open file specified by *handle*. The region to be locked or unlocked starts at the current file position, and is *length* bytes long.

Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

The *cmd* specifies the action to be taken (the values are defined in sys\locking.h):

LK_LOCK	Lock the region. If the lock is unsuccessful, try once a second for 10 seconds before giving up.
LK_RLCK	Same as LK_LOCK.
LK_NBLCK	Lock the region. If the lock if unsuccessful, give up immediately.
LK_NBRLCK	Same as LK_NBLCK.
LK_UNLCK	Unlock the region, which must have been previously locked.

Return Value

On successful operations, *locking* returns 0. Otherwise, it returns -1, and the global variable errno is set to one of the following values:

EACCES	File already locked or unlocked
EBADF	Bad file number
EDEADLOCK	File cannot be locked after 10 retries (*cmd* is LK_LOCK or LK_RLCK)
EINVAL	Invalid *cmd*, or SHARE.EXE not loaded

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

log, logl

See also Example Portability

Syntax

```
#include <math.h>
double log(double x);
long double logl(long double x);
```

Description

Calculates the natural logarithm of *x*.

log calculates the natural logarithm of *x*.

logl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with bcd and complex types.

Return Value

On success, *log* and *logl* return the value calculated, *ln(x)*.

If the argument *x* passed to these functions is real and less than 0, the global variable errno is set to

EDOM	Domain error

If *x* is 0, the functions return the value negative HUGE_VAL (*log*) or negative _LHUGE_VAL (*logl*), and set errno to ERANGE. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
log	+	+	+	+	+	+	+
logl	+		+	+			+

log10, log10l

See also Example Portability

Syntax

```
#include <math.h>
double log10(double x);
long double log10l(long double x);
```

Description

log10 calculates the base 10 logarithm of *x*.

log10l is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

Return Value

On success, *log10* (or *log10l*) returns the value calculated, $\log_{10}(x)$.

If the argument *x* passed to these functions is real and less than 0, the global variable errno is set to

EDOM Domain error

If *x* is 0, these functions return the value negative HUGE_VAL (*log10*) or _LHUGE_VAL (*log10l*). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
log10	+	+	+	+	+	+	+
log10l	+		+	+			+

longjmp

See also Example Portability

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf jmpb, int retval);
```

Description

Performs nonlocal goto.

A call to *longjmp* restores the task state captured by the last call to setjmp with the argument *jmpb*. It then returns in such a way that *setjmp* appears to have returned with the value *retval*.

A task state includes

Win 16	Win 32
All segment registers CS, DS, ES, SS	No segment registers are saved
Register variables	Register variables
DI and SI	EBX, EDI, ESI
Stack pointer SP	Stack pointer ESP
Frame pointer BP	Frame pointer EBP
Flags	Flags are not saved

A task state is complete enough that *setjmp* and *longjmp* can be used to implement co-routines.

setjmp must be called before *longjmp*. The routine that called *setjmp* and set up *jmpb* must still be active and cannot have returned before the *longjmp* is called. If this happens, the results are unpredictable.

longjmp cannot pass the value 0; if 0 is passed in *retval*, *longjmp* will substitute 1.

DOS Users

You cannot use *setjmp* and *longjmp* for implementing co-routines if your program is overlaid. Normally, *setjmp* and *longjmp* save and restore all the registers needed for co-routines, but the overlay manager needs to keep track of stack contents and assumes there is only one stack. When you implement co-routines, there are usually either two stacks or two partitions of one stack, and the overlay manager will not track them properly.

You can have background tasks that run with their own stacks or sections of stack, but you must ensure that the background tasks do not invoke any overlaid code, and you must not use the overlay versions of *setjmp* or *longjmp* to switch to and from background.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

lowvideo

See also Example Portability

Syntax

```
#include <conio.h>
void lowvideo(void);
```

Description

Selects low-intensity characters.

lowvideo selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen. It affects only those characters displayed by functions that perform text mode, direct console output *after* this function is called.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_lrotl, _lrotr

See also Example Portability

Syntax

```
#include <stdlib.h>
unsigned long _lrotl(unsigned long val, int count);
unsigned long _lrotr(unsigned long val, int count);
```

Description

Rotates an **unsigned long** integer value to the left or right.

*_lrotl*rotates the given *val* to the left *count* bits. *_lrotr* rotates the given *val* to the right *count* bits.

Return Value

The functions return the rotated integer:

- *_lrotl* returns the value of *val* left-rotated *count* bits.
- *_lrotr* returns the value of *val* right-rotated *count* bits.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

lsearch

See also Example Portability

Syntax

```
#include <stdlib.h>
void *lsearch(const void *key, void *base, size_t *num, size_t width, int
  (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lsearch searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to *lsearch*. If the item that *key* points to is not in the table, *lsearch* appends that item to the table.

- *base* points to the base (0th element) of the search table.
- *num* points to an integer containing the number of entries in the table.
- *width* contains the number of bytes in each entry.
- *key* points to the item to be searched for (the *search key*).

The function *fcmp* must be used with the _USERENTRY calling convention.

The argument *fcmp* points to a user-written comparison routine, that compares two items and returns a value based on the comparison.

To search the table, *lsearch* makes repeated calls to the routine whose address is passed in *fcmp*.

On each call to the comparison routine, *lsearch* passes two arguments:

key a pointer to the item being searched for

elem pointer to the element of *base* being compared.

fcmp is free to interpret the search key and the table entries in any way.

Return Value

lsearch returns the address of the first entry in the table that matches the search key.

If the search key is not identical to **elem*, *fcmp* returns a nonzero integer. If the search key is identical to **elem*, *fcmp* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

lseek

See also Example Portability

Syntax

```
#include <io.h>
long lseek(int handle, long offset, int fromwhere);
```

Description

Moves file pointer.

lseek sets the file pointer associated with *handle* to a new position *offset* bytes beyond the file location given by *fromwhere*. *fromwhere* must be one of the following symbolic constants (defined in io.h):

fromwhere	File location
SEEK_CUR	Current file pointer position
SEEK_END	End-of-file
SEEK_SET	File beginning

Return Value

lseek returns the offset of the pointer's new position measured in bytes from the file beginning. *lseek* returns -1L on error, and the global variable errno is set to one of the following values:

EBADF	Bad file handle
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_ltoa, _ltow, _i64toa, _ui64toa, _i64tow, _ui64tow

See also Example Portability

Syntax

```
#include <stdlib.h>
char *_ltoa(long value, char *string, int radix);
char *_i64toa(__int64 value, char *strP, int radix);
char *_ui64toa(unsigned __int64 value, char *strP, int radix);

// The following are Unicode versions
wchar_t *_ltow(long value, wchar_t *string, int radix);
wchar_t *_i64tow(__int64 value, wchar_t *strP, int radix);
wchar_t *_ui64tow(unsigned __int64 value, wchar_t *strP, int radix);
```

Description

Converts a **long** to a string.

_ltoa converts *value* to a null-terminated string and stores the result in *string*. *value* is a long integer.

radix specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (**-**).

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). ***_ltoa*** can return up to 33 bytes.

Return Value

_ltoa returns a pointer to *string*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_makepath, _wmakepath

See also Example Portability

Syntax

```
#include <stdlib.h>
void _makepath(char *path, const char *drive, const char *dir, const char
  *name, const char *ext);
void _wmakepath(wchar_t *path, const wchar_t *drive, const wchar_t *dir,
  const wchar_t *name, const wchar_t *ext);
```

Description

Builds a path from component parts.

_makepath makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

drive	=	X:
dir	=	\DIR\SUBDIR\
name	=	NAME
ext	=	.EXT

If *drive* is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If *dir* is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If *name* is empty or NULL, no file name is inserted in the path name.

If *ext* is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

_makepath assumes there is enough space in *path* for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.

_makepath and *_splitpath* are invertible; if you split a given *path* with *_splitpath*, then merge the resultant components with *_makepath*, you end up with *path*.

Return Value

None

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

malloc

See also Example Portability

Syntax

```
#include <stdlib.h> or #include<alloc.h>
void *malloc(size_t size);
```

Description

malloc allocates a block of *size* bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed, and in the exact amounts needed.

Allocates main memory.The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ heap memory allocation.

For 16-bit programs, all the space between the end of the data segment and the top of the program stack is available for use in the small data models, except for a small margin immediately before the top of the stack. This margin is intended to allow the application some room to make the stack larger, in addition to a small amount needed by DOS.

In the large data models, all the space beyond the program stack to the end of available memory is available for the heap.

Return Value

On success, *malloc* returns a pointer to the newly allocated block of memory. If not enough space exists for the new block, it returns NULL. The contents of the block are left unchanged. If the argument *size* == 0, *malloc* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_matherr, _matherrl

Example Portability

Syntax

```
#include <math.h>
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

Description

User-modifiable math error handler.

_matherr is called when an error is generated by the math library.

_matherrl is the **long double** version; it is called when an error is generated by the *long double* math functions.

_matherr and *_matherrl* each serve as a user hook (a function that can be customized by the user) that you can replace by writing your own math error-handling routine.

_matherr and *_matherrl* are useful for information on trapping domain and range errors caused by the math functions. They do not trap floating-point exceptions, such as division by zero. See *signal* for information on trapping such errors.

You can define your own *_matherr* or *_matherrl* routine to be a custom error handler (such as one that catches and resolves certain types of errors); this customized function overrides the default version in the C library. The customized *_matherr* or *_matherrl* should return 0 if it fails to resolve the error, or nonzero if the error is resolved. When *_matherr* or *_matherrl* return nonzero, no error message is printed and the global variable errno is not changed.

Here are the *_exception* and *_exceptionl* structures (defined in math.h):

```
struct _exception {
   int   type;
   char  *name;
   double arg1, arg2, retval;
};

struct _exceptionl {
   int   type;
   char  *name;
   long double arg1, arg2, retval;
};
```

The members of the *_exception* and *_exceptionl* structures are shown in the following table:

Member	What It Is (Or Represents)
type	The type of mathematical error that occurred; an enum type defined in the typedef _mexcep (see definition after this list).
name	A pointer to a null-terminated string holding the name of the math library function that resulted in an error.
arg1, arg2	The arguments (passed to the function that name points to) caused the error; if only one argument was passed to the function, it is stored in arg1.
retval	The default return value for _matherr (or _matherrl); you can modify this value.

The **typedef** _mexcep, also defined in math.h, enumerates the following symbolic constants representing possible mathematical errors:

Symbolic Constant	Mathematical Error
DOMAIN	Argument was not in domain of function, such as log(-1).
SING	Argument would result in a singularity, such as pow(0, -2).

OVERFLOW	Argument would produce a function result greater than DBL_MAX (or LDBL_MAX), such as exp(1000).
UNDERFLOW	Argument would produce a function result less than DBL_MIN (or LDBL_MIN), such as exp(-1000).
TLOSS	Argument would produce function result with total loss of significant digits, such as sin(10e70).

The macros DBL_MAX, DBL_MIN, LDBL_MAX, and LDBL_MIN are defined in float.h

The source code to the default *_matherr* and *_matherrl* is on the Borland C++ distribution disks.

The UNIX-style *_matherr* and *_matherrl* default behavior (printing a message and terminating) is not ANSI compatible. If you want a UNIX-style version of these routines, use MATHERR.C and MATHERRL.C provided on the Borland C++ distribution disks.

Return Value

The default return value for *_matherr* and *_matherrl* is 1 if the error is UNDERFLOW or TLOSS, 0 otherwise. *_matherr* and *_matherrl* can also modify *e -> retval*, which propagates back to the original caller.

When *_matherr* and *_matherrl* return 0 (indicating that they were not able to resolve the error), the global variable errno is set to 0 and an error message is printed.

When _matherr and _matherrl return nonzero (indicating that they were able to resolve the error), the global variable errno is not set and no messages are printed.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

max

See also Example Portability

Syntax

```
#include <stdlib.h> /* macro version */
(type) max(a, b);
template <class T> T max( T t1, T t2 ); // C++ only
```

Description

Returns the larger of two values.

The C macro and the C++ template function compare two values and return the larger of the two. Both arguments and the routine declaration must be of the same type.

Return Value

max returns the larger of two values.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_mbbtype

See also

Syntax

```
#include <mbstring.h>
int _mbbtype(unsigned char ch, int mode);
```

Description

The *_mbbtype* function inspects the multibyte argument, character *ch*, to determine whether it is a single-byte character, or whether *ch* is the leadbyte or trailing byte in a multibyte character. The *_mbbtype* function can determine whether *ch* is an invalid character.

Return Value

The value that *_mbbtype* returns is one of the following manifest constants, defined in mbctype.h. The return value depends on the value of *ch* and the test which you want performed on *ch*.

Value of *mode*	Value of *ch*	Test performed	Return value
mode != 1	Single byte	Valid single or lead byte	**_MBC_SINGLE**
mode != 1	Leadbyte	Valid single or lead byte	**_MBC_LEAD**
mode = 1	Trailbyte	Valid single or trail byte	**_MBC_TRAIL**
Any value	Any value	Valid character	**_MBC_ILLEGAL**

_mbccpy

Syntax

```
#include <mbstring.h>
void _mbccpy(unsigned char *dest, unsigned char *src);
```

Description

The *_mbccpy* function copies a multibyte character from *src* to *dest*. The *_mbccpy* function makes an implicit call to *_ismbblead* so that the *src* pointer references a lead byte. If *src* doesn't reference a lead byte, no copy is performed.

Return Value

None.

mblen

See also Example Portability

Syntax

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

Determines the length of a multibyte character.

If *s* is not null, *mblen* determines the number of bytes in the multibyte character pointed to by *s*. The maximum number of bytes examined is specified by *n*.

The behavior of *mblen* is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If *s* is null, *mblen* returns a nonzero value if multibyte characters have state-dependent encodings. Otherwise, *mblen* returns 0.

If *s* is not null, *mblen* returns 0 if *s* points to the null character, and -1 if the next *n* bytes do not comprise a valid multibyte character; the number of bytes that comprise a valid multibyte character.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

_mbsbtype

See also

Syntax

```
#include <mbstring.h>
int _mbsbtype(const unsigned char *str, size_t nbyte);
```

Description

The *nbyte* argument specifies the number of bytes from the start of the zero-based string.

The *_mbsbtype* function inspects the argument *str* to determine whether the byte at the position specified by *nbyte* is a single-byte character, or whether it is the leadbyte or trailing byte in a multibyte character. The *_mbsbtype* function can determine whether the byte pointed at is an invalid character or a NULL byte.

Any invalid bytes in *str* before *nbyte* are ignored.

Return Value

The value that *_mbsbtype* returns is one of the following manifest constants, defined in mbctype.h.

Type of byte found	Return value
Single byte	**_MBC_SINGLE**
Leadbyte	**_MBC_LEAD**
Trailbyte	**_MBC_TRAIL**
Invalid character or byte	**_MBC_ILLEGAL**

_mbsnbcmp

See also

Syntax

```
#include <mbstring.h>
int _mbsnbcmp(const unsigned char *s1, const unsigned char s2, size_t
  maxlen);
```

Description

_mbsnbcmp makes an case-sensitive comparison of *s1* and *s2* for no more than *maxlen* bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined *maxlen* bytes.

_mbsnbcmp is case sensitive.

_mbsnbcmp is not affected by locale.

_mbsnbcmp compares bytes based on the current multibyte code page.

Return Value

_mbsnbcmp returns an integer value based on the result of comparing *s1* (or part of it) to *s2* (or part of it):

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

_mbsnbcoll, _mbsnbicoll

See also

Syntax

```
#include <mbstring.h>
int _mbsnbcoll(const unsigned char *s1, const unsigned char *s2, maxlen);
int _mbsnbicoll(const unsigned char *s1, const unsigned char *s2, maxlen);
```

Description

_mbsnbicoll is the case-insensitive version of *_mbsnbcoll*.

These functions collate the strings specified by arguments *s1* and *s2*. The collation order is determined by lexicographic order as specified by the current multibyte code page. At most, *maxlen* number of bytes are collated.

Note: The lexicographic order is not always the same as the order of characters in the character set.

If the last byte in *s1* or *s2* is a leadbyte, it is not compared.

Return Value

Each of these functions return an integer value based on the result of comparing *s1* (or part of it) to *s2* (or part of it):

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

On error, each of these functions returns **_NLSCMPERROR**.

_mbsnbcpy

See also

Syntax

```
#include <mbstring.h>
unsigned char *_mbsnbcpy(unsigned char *dest, unsigned char *src, size_t
  maxlen);
```

Description

The *_mbsnbcpy* function copies at most *maxlen* number of characters from the *src* buffer to the *dest* buffer. The *dest* buffer is null terminated after the copy.

It is the user's responsibility to be sure that *dest* is large enough to allow the copy. An improper buffer size can result in memory corruption.

Return Value

The functon returns *dest*.

_mbsnbicmp

See also

Syntax

```
#include <mbstring.h>
int _mbsnbicmp(const unsigned char *s1, const unsigned char s2, size_t
  maxlen);
```

Description

_mbsnbicmp ignores case while making a comparison of *s1* and *s2* for no more than *maxlen* bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined *maxlen* bytes.

_mbsnbicmp is not case sensitive.

_mbsnbicmp is not affected by locale.

_mbsnbicmp compares bytes based on the current multibyte code page.

Return Value

_mbsnbicmp returns an integer value based on the result of comparing *s1* (or part of it) to *s2* (or part of it):

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

_mbsnbset

See also

Syntax

```
#include <mbstring.h>
unsigned char *_mbsnbset(unsigned char str, unsigned int ch, size_t maxlen);
```

Description

_mbsnbset sets at most *maxlen* number of bytes in the string *str* to the character *ch*. The argument *ch* can be a single or multibyte character.

The function quits if the terminating null character is found before *maxlen* is reached. If *ch* is a multibyte character that cannot be accomodated at the end of *str*, the last character in *str* is set to a blank character.

Return Value

strset returns *str*.

_mbsninc, _strninc, _wcsninc

See also

Syntax

```
#include <mbstring.h>
unsigned char *_mbsninc(const unsigned char *str, size_t num);
```

Description

These functions should be accessed throught the portable macro, *_tcsninc*, defined in tchar.h.

The functions increment the character array *str* by *num* number of characters.

Return value

The functions return a pointer to the resized character string specified by the argument *str*.

_mbsnbcnt, _mbsnccnt, _strncnt, _wcsncnt

See also

Syntax

```
#include <mbstring.h>
size_t  _mbsnbcnt(const unsigned char * str, size_t nmbc);
size_t  _mbsnccnt(const unsigned char * str, size_t nbyte);
```

Description

If _MBCS is defined:

- *_mbsnbcnt* is mapped to the portable macro *_tcsnbcnt*
- *_mbsnccnt* is mapped to the portable macro *_tcsnccnt*

If _UNICODE is defined:

- both *_mbsnbcnt* and *_mbsnccnt* are mapped to the *_wcsncnt* macro

If neither _MBCS nor _UNICODE are defined.

- *_tcsnbcnt* and *_tcsnccnt* are mapped to the *_strncnt* macro

_strncnt is the single-byte version of these functions.

_wcsncnt is the wide-character version of these functions.

_strncnt and *_wcsncnt* are available only for generic-text mappings. They should not be used directly.

_mbsnbcnt examines the first *nmbc* multibyte characters of the *str* argument. The function returns the number of bytes found in the those characters.

_mbsnccnt examines the first *nmbc* bytes of the *str* argument. The function returns the number of characters found in those bytes. If NULL is encountered in the second byte of a multibyte character, the whole character is considered NULL and will not be included in the return value.

Each of the functions ends its examination of the *str* argument if NULL is reached before the specified number of characters or bytes is examined.

If *str* has fewer than the specified number of characters or bytes, the function return the number of characters or bytes found in *str*.

Return Value

_mbsnbcnt returns the number of bytes found.

_mbsnccnt returns the number of characters found.

If *nmbc* or *nbyte* are less than zero, the functions return 0.

_mbsspnp, _strspnp, _wcsspnp

See also Example

Syntax

```
#include <mbstring.h>
unsigned char *_mbsspnp(const unsigned char *s1, const unsigned char *s2);
```

Description

Use the portable macro, *_tcsspnp*, defined in tchar.h, to access these functions.

Each of these functions search for the first character in *s1* that is not contained in *s2*.

Return Value

The functions return a pointer to the first character in *s1* that is not found in the character set for *s2*.

If every character from *s1* is found in *s2*, each of the functions return NULL.

mbstowcs

See also Example Portability

Syntax

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

Converts a multibyte string to a **wchar_t** array.

The function converts the multibyte string *s* into the array pointed to by *pwcs*. No more than *n* values are stored in the array. If an invalid multibyte sequence is encountered, *mbstowcs* returns (*size_t*) -1.

The *pwcs* array will not be terminated with a zero value if *mbstowcs* returns *n*.

Return Value

If an invalid multibyte sequence is encountered, *mbstowcs* returns (*size_t*) -1. Otherwise, the function returns the number of array elements modified, not including the terminating code, if any.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

mbtowc

See also Example Portability

Syntax

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

Converts a multibyte character to **wchar_t** code.

If *s* is not null, *mbtowc* determines the number of bytes that comprise the multibyte character pointed to by *s*. Next, *mbtowc* determines the value of the type **wchar_t** that corresponds to that multibyte character. If there is a successful match between **wchar_t** and the multibyte character, and *pwc* is not null, the **wchar_t** value is stored in the array pointed to by *pwc*. At most *n* characters are examined.

Return Value

When *s* points to an invalid multibyte character, -1 is returned. When *s* points to the null character, 0 is returned. Otherwise, *mbtowc* returns the number of bytes that comprise the converted multibyte character.

The return value never exceeds MB_CUR_MAX or the value of *n*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

memccpy, _fmemccpy

See also Example Portability

Syntax

```
#include <mem.h>
void *memccpy(void *dest, const void *src, int c, size_t n);
void far * far _fmemccpy(void far *dest, const void far *src, int c, size_t
  n)
```

Description

Copies a block of *n* bytes.

memccpy is available on UNIX System V systems.

memccpy copies a block of *n* bytes from *src* to *dest*. The copying stops as soon as either of the following occurs:

- The character *c* is first copied into *dest*.
- *n* bytes have been copied into *dest*.

Return Value

memccpy returns a pointer to the byte in *dest* immediately following *c*, if *c* was copied; otherwise, *memccpy* returns NULL.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memccpy	+	+	+	+			+
_fmemccpy	+		+				

memchr, _wmemchr, _fmemchr

Example Portability

Syntax

```
#include <mem.h>
void *memchr(const void *s, int c, size_t n);                        /* C
  only */
void far * far _fmemchr(const void far *s, int c, size_t n);         /* C
  only */

const void *memchr(const void *s, int c, size_t n);                   // C++
  only
void *memchr(void *s, int c, size_t n);                               // C++
  only
const void far * far _fmemchr(const void far *s, int c, size_t n);    //C++
  only
void far * far _fmemchr(void far *s, int c, size_t n);       // C++ only
void *memchr(const void *s, int c, size_t n);
void * _wmemchr(void *s, int c, size_t n);             // Unicode version
void far * far _fmemchr(const void far *s, int c, size_t n);
```

Description

Searches *n* bytes for character *c*.

memchr is available on UNIX System V systems.

memchr searches the first *n* bytes of the block pointed to by *s* for character *c*.

Return Value

On success, *memchr* returns a pointer to the first occurrence of *c* in *s*; otherwise, it returns NULL.

Note: If you are using the intrinsic version of these functions, the case of *n* = 0 will return NULL.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memchr	+	+	+	+	+	+	+
_fmemchr	+		+				

memcmp, _fmemcmp

See also Example Portability

Syntax

```
#include <mem.h>
int memcmp(const void *s1, const void *s2, size_t n);
int far _fmemcmp(const void far *s1, const void far *s2, size_t n)
```

Description

Compares two blocks for a length of exactly *n* bytes.

memcmp is available on UNIX System V systems.

memcmp compares the first *n* bytes of the blocks *s1* and *s2* as **unsigned char**s.

Return Value

Because it compares bytes as **unsigned char**s, *memcmp* returns a value that is

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

For example,

```
memcmp("\xFF", "\x7F", 1)
```

returns a value greater than 0.

Note: If you are using the intrinsic version of these functions, the case of *n* = 0 will return NULL.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memcmp	+	+	+	+	+	+	+
_fmemcmp	+		+				

memcpy, _wmemcpy, _fmemcpy

See also Example Portability

Syntax

```
#include <mem.h>
void *memcpy(void *dest, const void *src, size_t n);
void *_wmemcpy(void *dest, const void *src, size_t n);
void far *far _fmemcpy(void far *dest, const void far *src, size_t n);
```

Description

Copies a block of *n* bytes.

memcpy is available on UNIX System V systems.

memcpy copies a block of *n* bytes from *src* to *dest*. If *src* and *dest* overlap, the behavior of *memcpy* is undefined.

Return Value

memcpy returns *dest*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memcpy	+	+	+	+	+	+	+
_fmemcpy	+		+				

memicmp, _fmemicmp

See also Example Portability

Syntax

```
#include <mem.h>
int memicmp(const void *s1, const void *s2, size_t n);
int far _fmemicmp(const void far *s1, const void far *s2, size_t n)
```

Description

Compares *n* bytes of two character arrays, ignoring case.

memicmp is available on UNIX System V systems.

memicmp compares the first *n* bytes of the blocks *s1* and *s2*, ignoring character case (upper or lower).

Return Value

memicmp returns a value that is

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memicmp	+	+	+	+			+
_fmemicmp	+		+				

memmove, _fmemmove

See also Example Portability

Syntax

```
#include <mem.h>
void *memmove(void *dest, const void *src, size_t n);
void far * far _fmemmove (void far *dest, const void far *src, size_t n)
```

Description

Copies a block of *n* bytes.

memmove copies a block of *n* bytes from *src* to *dest*. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly.

_fmemmove is the far version.

Return Value

memmove and *_fmemmove* return *dest*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memmove	+	+	+	+	+	+	+
_fmemmove	+		+				

memset, _wmemset, _fmemset

See also Example Portability

Syntax

```
#include <mem.h>
void *memset(void *s, int c, size_t n);
void *_wmemset(void *s, int c, size_t n);
void far * far _fmemset (void far *s, int c, size_t n)
```

Description

Sets *n* bytes of a block of memory to byte *c*.

memset sets the first *n* bytes of the array *s* to the character *c*.

Return Value

memset returns *s*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memset	+	+	+	+	+	+	+
_fmemset	+		+				

min

See also Example Portability

Syntax

```
#include <stdlib.h> /* macro version */
(type) min(a, b);
template <class T> T min( T t1, T t2 ); // C++ only
```

Description

Returns the smaller of two values.

The C macro and the C++ template function compare two values and return the smaller of the two. Both arguments and the routine declaration must be of the same type.

Return Value

min returns the smaller of two values.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

mkdir, _wmkdir

See also Example Portability

Syntax

```
#include <dir.h>
int mkdir(const char *path);
int _wmkdir(const wchar_t *path);
```

Description

Creates a directory.

mkdir is available on UNIX, though it then takes an additional parameter.

mkdir creates a new directory from the given path name *path*.

Return Value

mkdir returns the value 0 if the new directory was created.

A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

MK_FP

See also Example Portability

Syntax

```
#include <dos.h>
void far * MK_FP(unsigned seg, unsigned ofs);
```

Description

Makes a **far** pointer.

MK_FP is a macro that makes a **far** pointer from its component segment (*seg*) and offset (*ofs*) parts.

Return Value

MK_FP returns a **far** pointer.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_mktemp, _wmktemp

Example Portability

Syntax

```
#include <dir.h>
char *_mktemp(char *template);
wchar_t *_wmktemp(wchar_t *template);
```

Description

Makes a unique file name.

_mktemp replaces the string pointed to by *template* with a unique file name and returns *template*.

template should be a null-terminated string with six trailing Xs. These Xs are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.

Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same format.

Return Value

If a unique name can be created and *template* is well-formed, *_mktemp* returns the address of the *template* string. Otherwise, it returns null.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

mktime

See also Example Portability

Syntax

```
#include <time.h>
time_t mktime(struct tm *t);
```

Description

Converts time to calendar format.

Converts the time in the structure pointed to by *t* into a calendar time with the same format used by the *time* function. The original values of the fields *tm_sec, tm_min, tm_hour, tm_mday*, and *tm_mon* are not restricted to the ranges described in the *tm* structure. If the fields are not in their proper ranges, they are adjusted. Values for fields *tm_wday* and *tm_yday* are computed after the other fields have been adjusted.

The allowable range of calendar times is Jan 1 1970 00:00:00 to Jan 19 2038 03:14:07.

Return Value

On success, *mktime* returns calendar time as described above.

On error (if the calendar time cannot be represented), *mktime* returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

modf, modfl

See also Example Portability

Syntax

```
#include <math.h>
double modf(double x, double *ipart);
long double modfl(long double x, long double *ipart);
```

Description

Splits a **double** or **long double** into integer and fractional parts.

modf breaks the **double** *x* into two parts: the integer and the fraction. *modf* stores the integer in *ipart* and returns the fraction.

modfl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

modf and *modfl* return the fractional part of *x*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
modf	+	+	+	+	+	+	+
modfl	+		+	+			+

movedata

See also Example Portability

Syntax

```
#include <mem.h>
void movedata(unsigned srcseg, unsigned srcoff, unsigned dstseg, unsigned
  dstoff, size_t n);
```

Description

Copies *n* bytes.

movedata copies *n* bytes from the source address (*srcseg:srcoff*) to the destination address (*dstseg:dstoff*). *movedata* provides a memory-model independent means for moving blocks of data.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

movmem, _fmovmem

See also Examples Portability

Syntax

```
#include <mem.h>
void movmem(const void *src, void *dest, unsigned length);
void _fmovmem(const void far *src, void far *dest, unsigned length);
```

Description

Moves a block of *length* bytes.

movmem moves a block of *length* bytes from *src* to *dest*. Even if the source and destination blocks overlap, the move direction is chosen so that the data is always moved correctly.

_fmovmem is the far version.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

movetext

See also Example Portability

Syntax

```
#include <conio.h>
int movetext(int left, int top, int right, int bottom, int destleft, int
  desttop);
```

Description

Copies text onscreen from one rectangle to another.

movetext copies the contents of the onscreen rectangle defined by *left, top, right*, and *bottom* to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (*destleft, desttop*).

All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly.

movetext is a text mode function performing direct video output.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

On success, *movetext* returns nonzero.

On error (for example, if it failed because you gave coordinates outside the range of the current screen mode), *movetext* returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_msize

See also Example Portability

Syntax

```
#include <malloc.h>
size_t _msize(void *block);
```

Description

Returns the size of a heap block.

_msize returns the size of the allocated heap block whose address is *block*. The block must have been allocated with *malloc, calloc*, or *realloc*. The returned size can be larger than the number of bytes originally requested when the block was allocated.

Return Value

_msize returns the size of the block in bytes.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

normvideo

See also Example Portability

Syntax

```
#include <conio.h>
void normvideo(void);
```

Description

Selects normal-intensity characters.

normvideo selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

This function does not affect any characters currently on the screen, only those displayed by functions (such as *cprintf*) performing direct console output functions after *normvideo* is called.

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

offsetof

Example Portability

Syntax

```
#include <stddef.h>
size_t offsetof(struct_type, struct_member);
```

Description

Gets the byte offset to a structure member.

offsetof is available only as a macro. The argument *struct_type* is a **struct** type. *struct_member* is any element of the **struct** that can be accessed through the member selection operators or pointers.

If *struct_member* is a bit field, the result is undefined.

See also Chapter 2 in the *Programmer's Guide* for a discussion of the **sizeof** operator, memory allocation, and alignment of structures.

Return Value

offsetof returns the number of bytes from the start of the structure to the start of the named structure member.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

open, _wopen

See also Example Portability

Syntax

```
#include <fcntl.h>
#include<io.h>
int open(const char *path, int access [, unsigned mode]);
int _wopen(const wchar_t *path, int access [, unsigned mode]);
```

Description

Opens a file for reading or writing.

open opens the file specified by *path*, then prepares it for reading and/or writing as determined by the value of *access*.

To create a file in a particular mode, you can either assign to the global variable *_fmode* or call *open* with the O_CREAT and O_TRUNC options ORed with the translation mode desired.

For example, the call

```
open("XMP",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

creates a binary-mode, read-only file named XMP, truncating its length to 0 bytes if it already existed.

For *open*, *access* is constructed by bitwise ORing flags from the following lists. Only one flag from the first list can be used (and one *must* be used); the remaining flags can be used in any logical combination.

These symbolic constants are defined in fcntl.h.

Read/Write Flags

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Other Access Flags

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits as in *chmod*.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	Can be given to explicitly open the file in binary mode.
O_TEXT	Can be given to explicitly open the file in text mode.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing access, you need to supply the *mode* argument to *open* from the following symbolic constants defined in sys\stat.h.

Value Of Mode	Access Permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

On success, *open* returns a nonnegative integer (the file handle). The file pointer, which marks the

current position in the file, is set to the beginning of the file.

On error, *open* returns -1 and the global variable errno is set to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	No such file or directory

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

opendir, wopendir

See also Example Portability

Syntax

```
#include <dirent.h>
DIR *opendir(char *dirname);
wDIR *wopendir(const wchar_t *dirname);
```

Description

Opens a directory stream for reading.

opendir is available on POSIX-compliant UNIX systems.

The *opendir* function opens a directory stream for reading. The name of the directory to read is *dirname*. The stream is set to read the first entry in the directory.

A directory stream is represented by the *DIR* structure, defined in dirent.h. This structure contains no user-accessible fields. Multiple directory streams can be opened and read simultaneously. Directory entries can be created or deleted while a directory stream is being read.

Use the *readdir* function to read successive entries from a directory stream. Use the *closedir* function to remove a directory stream when it is no longer needed.

Return Value

On success, *opendir* returns a pointer to a directory stream that can be used in calls to readdir, rewinddir, and closedir.

On errror (If the directory cannot be opened), it returns NULL and sets the global variable errno to

ENOENT	The directory does not exist
ENOMEM	Not enough memory to allocate a DIR object

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_open_osfhandle

See also Example Portability

Syntax

```
#include <io.h>
int _open_osfhandle(long osfhandle, int flags);
```

Description

Associates file handles.

The _open_osfhandle function allocates a run-time file handle and sets it to point to the operating system file handle specified by *osfhandle*.

The value flags is a bitwise OR combination of one or more of the following manifest constants (defined in fcntl.h):

O_APPEND	Repositions the file pointer to the end of the file before every write operation.
O_RDONLY	Opens the file for reading only.
O_TEXT	Opens the file in text (translated) mode.

Return Value

On success, _open_osfhandle returns a C run-time file handle. Otherwise, it returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			

outp

See also Example Portability

Syntax

```
#include <conio.h>
int outp(unsigned portid, int value);
```

Description

Outputs a byte to a hardware port.

outp is a macro that writes the low byte of *value* to the output port specified by *portid*.

If *outp* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and #undef the macro *outp*, you'll get the *outp* function.

Return Value

outp returns *value*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

outport, outportb

See also Examples Portability

Syntax

```
#include <dos.h>
void outport(int portid, int value);
void outportb(int portid, unsigned char value);
```

Description

Outputs a word or byte to a hardware port.

outport works just like the 80x86 instruction OUT. It writes the low byte of the word given by *value* to the output port specified by *portid* and writes the high byte of the word to *portid* +1.

outportb is a macro that writes the byte given by *value* to the output port specified by *portid*.

If you include dos.h, *outportb* will be treated as a macro that expands to inline code. If you do not include dos.h, or if you include dos.h and #undef the macro *outportb*, you will get the *outportb* function.

Return Value

None.

Examples

outport

outportb

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

outpw

See also Example Portability

Syntax

```
#include <conio.h>
unsigned outpw(unsigned portid, unsigned value);
```

Description

Outputs a word to a hardware port.

outpw is a macro that writes the 16-bit word given by *value* to the output port specified by *portid*. It writes the low byte of *value* to *portid*, and the high byte of the word to *portid* +1, using a single 16-bit *OUT* instruction.

If *outpw* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and #undef the macro *outpw*, you'll get the *outpw* function.

Return Value

outpw returns *value*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

parsfnm

Example Portability

Syntax

```
#include <dos.h>
char *parsfnm(const char *cmdline, struct fcb *fcb, int opt);
```

Description

Parses file name.

parsfnm parses a string pointed to by *cmdline* for a file name. The string is normally a command line. The file name is placed in a file control block (FCB) as a drive, file name, and extension. The FCB is pointed to by *fcb*.

The *opt* parameter is the value documented for AL in the DOS parse system call. See your DOS reference manuals under system call 0x29 for a description of the parsing operations performed on the file name.

Return Value

On success, *parsfnm* returns a pointer to the next byte after the end of the file name.

On error (in parsing the file name), *parsfnm* returns null.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_pclose

See also Example Portability

Syntax

```
#include <stdio.h>
int _pclose(FILE * stream);
```

Description

Waits for piped command to complete.

_pclose closes a pipe stream created by a previous call to _popen, and then waits for the associated child command to complete.

Return Value

On success, *_pclose* returns the termination status of the child command. This is the same value as the termination status returned by cwait, except that the high and low order bytes of the low word are swapped.

On error, it returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

peek

See also Example Portability

Syntax

```
#include <dos.h>
int peek(unsigned segment, unsigned offset);
```

Description

Returns the word at memory location specified by *segment:offset*.

peek returns the word at the memory location *segment:offset*.

If *peek* is called when dos.h has been included, it is treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and #undef *peek*, you'll get the function rather than the macro.

Return Value

peek returns the word of data stored at the memory location *segment:offset*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

peekb

See also Example Portability

Syntax

```
#include <dos.h>
char peekb(unsigned segment, unsigned offset);
```

Description

Returns the byte of memory specified by *segment:offset*.

peekb returns the byte at the memory location addressed by *segment:offset*.

If *peekb* is called when dos.h has been included, it is treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and #undef *peekb*, you'll get the function rather than the macro.

Return Value

peekb returns the byte of information stored at the memory location *segment:offset*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

perror, _wperror

See also Example Portability

Syntax

```
#include <stdio.h>
void perror(const char *s);
void _wperror(const wchar_t *s);
```

Description

Prints a system error message.

perror prints to the *stderr* stream (normally the console) the system error message for the last library routine that set the global variable errno.

It prints the argument *s* followed by a colon (:) and the message corresponding to the current value of the global variable *errno* and finally a new line. The convention is to pass the file name of the program as the argument string.

The array of error message strings is accessed through the global variable _sys_errlist. The global variable *errno* can be used as an index into the array to find the string corresponding to the error number. None of the strings include a newline character.

The global variable _sys_nerr contains the number of entries in the array.

The following messages are generated by *perror*:

Win 16 and Win 32 messages

Arg list too big
Attempted to remove current directory
Bad address
Bad file number
Block device required
Broken pipe
Cross-device link
Error 0
Exec format error
Executable file in use
File already exists
File too large
Illegal seek
Inappropriate I/O control operation
Input/output error
Interrupted function call
Invalid access code
Invalid argument Resource busy
Invalid dataResource temporarily unavailable
Invalid environment
Invalid format
Invalid function number
Invalid memory block address
Is a directory
Math argument
Memory arena trashed

Name too long
No child processes
No more files
No space left on device
No such device
No such device or address
No such file or directory
No such process
Not a directory
Not enough memory
Not same device
Operation not permitted
Path not found
Permission denied
Possible deadlock
Read-only file system
Resource busy
Resource temporarily unavailable
Result too large
Too many links
Too many open files

Win 32 only messages

Note: For Win32s or Win32 GUI applications, stderr must be redirected.
Bad address
Block device required
Broken pipe
Executable file in use
File too large
Illegal seek
Inappropriate I/O control
Input/output error
Interrupted function call
Is a directory
Name too long
No child processes
No space left on device
No such device or address
No such process
Not a directory
Operation not permitted
Possible deadlock
Read-only file system
Resource busy
Resource temporarily unavailable
Too many links

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

_pipe

See also Example Portability

Syntax

```
#include <fcntl.h>
#include<io.h>
int _pipe(int *handles, unsigned int size, int mode);
```

Description

Creates a read/write pipe.

The *_pipe* function creates an anonymous pipe that can be used to pass information between processes. The pipe is opened for both reading and writing. Like a disk file, a pipe can be read from and written to, but it does not have a name or permanent storage associated with it; data written to and from the pipe exist only in a memory buffer managed by the operating system.

The read handle is returned to *handles*[0], and the write handle is returned to *handles*[1]. The program can use these handles in subsequent calls to read, write, dup, dup2, or close. When all pipe handles are closed, the pipe is destroyed.

The size of the internal pipe buffer is *size*. A recommended minimum value is 512 bytes.

The translation mode is specified by *mode*, as follows:

O_BINARY	The pipe is opened in binary mode
O_TEXT	The pipe is opened in text mode

If *mode* is zero, the translation mode is determined by the external variable *_fmode*.

Return Value

On success, *_pipe* returns 0 and returns the pipe handles to *handles*[0] and *handles*[1].

On error, it returns -1 and sets errno to one of the following values:

EMFILE	Too many open files
ENOMEM	Out of memory

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

poke

See also Example Portability

Syntax

```
#include <dos.h>
void poke(unsigned segment, unsigned offset, int value);
```

Description

Stores an integer value at a memory location given by *segment:offset*.

poke stores the integer *value* at the memory location *segment:offset*.

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and #undef *poke*, you'll get the function rather than the macro.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

pokeb

See also Example Portability

Syntax

```
#include <dos.h>
void pokeb(unsigned segment, unsigned offset, char value);
```

Description

Stores a byte value at memory location *segment:offset*.

pokeb stores the byte *value* at the memory location *segment:offset*.

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and #undef *pokeb*, you'll get the function rather than the macro.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

poly, polyl

Example Portability

Syntax

```
#include <math.h>
double poly(double x, int degree, double coeffs[]);
long double polyl(long double x, int degree, long double coeffs[]);
```

Description

Generates a polynomial from arguments.

poly generates a polynomial in *x*, of degree *degree*, with coefficients *coeffs[0]*, *coeffs[1], ..., coeffs[degree]*. For example, if *n* = 4, the generated polynomial is:

$$coeffs[4]x^4 + coeffs[3]x^3 + coeffs[2]x^2 + coeffs[1]x + coeffs[0]$$

polyl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

poly and *polyl* return the value of the polynomial as evaluated for the given *x*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
poly	+	+	+	+			+
polyl	+		+	+			+

_popen, _wpopen

See also Example Portability

Syntax

```
#include <stdio.h>
FILE *_popen (const char *command, const char *mode);
FILE *_wpopen (const wchar_t *command, const wchar_t *mode);
```

Description

Creates a command processor pipe.

The *_popen* function creates a pipe to the command processor. The command processor is executed asynchronously, and is passed the command line in *command*. The *mode* string specifies whether the pipe is connected to the command processor's standard input or output, and whether the pipe is to be opened in binary or text mode.

The *mode* string can take one of the following values:

Value	Description
rt	Read child command's standard output (text).
rb	Read child command's standard output (binary).
wt	Write to child command's standard input (text).
wb	Write to child command's standard input (binary).

The terminating *t* or *b* is optional; if missing, the translation mode is determined by the external variable *_fmode*.

Use the _pclose function to close the pipe and obtain the return code of the command.

Return Value

On success, *_popen* returns a FILE pointer that can be used to read the standard output of the command, or to write to the standard input of the command, depending on the *mode* string.

On error, it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

pow, powl

See also Example Portability

Syntax

```
#include <math.h>
double pow(double x, double y);
long double powl(long double x, long double y);
```

Description

Calculates *x* to the power of *y*.

powl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

This function can be used with bcd and complex types.

Return Value

On success, *pow* and powl return the value calculated of *x* to the power of *y*.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, the functions return the value HUGE_VAL (*pow*) or _LHUGE_VAL (*powl*). Results of excessively large magnitude can cause the global variable errno to be set to

ERANGE	Result out of range

If the argument *x* passed to *pow* or *powl* is real and less than 0, and *y* is not a whole number, or you call *pow(0,0)*, the global variable *errno* is set to

EDOM	Domain error

Error handling for these functions can be modified through the functions _matherr and _matherrl.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
pow	+	+	+	+	+	+	+
powl	+		+	+			+

pow10, pow10l

See also Example Portability

Syntax

```
#include <math.h>
double pow10(int p);
long double pow10l(int p);
```

Description

Calculates 10 to the power of *p*.

pow10l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, *pow10* returns the value calculated, 10 to the power of *p* and *pow10l* returns a **long double** result.

The result is actually calculated to **long double** accuracy. All arguments are valid, although some can cause an underflow or overflow.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
pow10	+	+	+	+			+
pow10l	+		+	+			+

printf, wprintf

See also Example Portability

Syntax

```
#include <stdio.h>
int printf(const char *format[, argument, ...]);
int wprintf(const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to stdout.

The *printf* function:

- Accepts a series of arguments
- Appllies to each argument a format specifier contained in the format string *format
- Outputs the formatted data (to the screen, a stream, *stdout*, or a string)

There must be enough arguments for the format. If there are not, the results will be unpredictable and likely disastrous. Excess arguments (more than required by the format) are merely ignored.

Note: For Win32s or Win32 GUI applications, *stdout* must be redirected.

Return Value

On success, *printf* returns the number of bytes output.

On error, *printf* returns EOF.

More About printf

Unicode output format specifiers
Format String
Format Specifiers
Format Specifier Conventions
Flag Characters
Input-size Modifiers
Precision Specifiers
Type Characters
Width Specifiers

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

printf Format String

See also

The format string, present in each of the *printf* function calls, controls how each function will convert, format, and print its arguments.

Note: There must be enough arguments for the format; if not, the results will be unpredictable and possibly disastrous. Excess arguments (more than required by the format) are ignored.

The format string is a character string that contains two types of objects:

- Plain characters are copied verbatim to the output stream.
- Conversion specifications fetch arguments from the argument list and apply formatting to them.

Plain characters are simply copied verbatim to the output stream.

Conversion specifications fetch arguments from the argument list and apply formatting to them.

printf Format Specifiers

See also

print format specifiers have the following form

```
% [flags] [width] [.prec] [F|N|h|l|L] type_char
```

Each format specifier begins with the percent character (`%`).

After the `%` come the following optional specifiers, in this order:

Optional Format String Components

These are the general aspects of output formatting controlled by the optional characters, specifiers, and modifiers in the format string:

Component	Optional/Required	What it Controls or Specifies
`[flags]`	(Optional)	Flag character(s) Output justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes
`[width]`	(Optional)	Width specifier Minimum number of characters to print, padding with blanks or zeros
`[prec]`	(Optional)	Precision specifier Maximum number of characters to print; for integers, minimum number of digits to print
`[F|N|h|l|L]`	(Optional)	Input size modifier Override default size of next input argument: **`N`** = **near** pointer **`F`** = **far** pointer **`h`** = **short int** **`l`** = **long** **`L`** = **long double**
`type_char`	(Required)	Conversion-type character

printf Flag characters

See also

They can appear in any order and combination.

Flag	What it means
-	Left-justifies the result, pads on the right with blanks. If not given, it right-justifies the result, pads on the left with zeros or blanks.
+	Signed conversion results always begin with a plus (+) or minus (-) sign.
blank	If value is nonnegative, the output begins with a blank instead of a plus; negative values still begin with a minus.
#	Specifies that *arg* is to be converted using an alternate form.

Note: Plus (+) takes precedence over blank () if both are given.

Alternate Forms for printf Conversion

See also

If you use the # flag conversion character, it has the following effect on the argument (*arg*) being converted:

Conversion character	How # affects the argument
c s d iu	No effect.
o	0 is prepended to a nonzero *arg*.
x X	0x (or 0X) is prepended to *arg*.
e E f	The result always contains a decimal point even if no digits follow the point. Normally, a decimal point appears in these results only if a digit follows it.
g G	Same as **e** and **E**, except that trailing zeros are not removed.

printf Width Specifiers

See also

The width specifier sets the minimum field width for an output value.

Width is specified in one of two ways:

- directly, through a decimal digit string
- indirectly, through an asterisk (*)

If you use an asterisk for the width specifier, the next argument in the call (which must be an **int**) specifies the minimum output field width.

Nonexistent or small field widths do *not* cause truncation of a field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Width specifier	How output width is affected
n	At least *n* characters are printed. If the output value has less than *n* characters, the output is padded with blanks (right-padded if - flag given, left-padded otherwise).
0*n*	At least *n* characters are printed. If the output value has less than *n* characters, it is filled on the left with zeros.
*	The argument list supplies the width specifier, which must precede the actual argument being formatted.

printf Precision Specifiers

See also

The *printf* precision specifiers set the maximum number of characters (or minimum number of integer digits) to print.

A *printf* precision specification always begins with a period (.) to separate it from any preceding width specifier.

Then, like `[width]`, precision is specified in one of two ways:

- directly, through a decimal digit string
- indirectly, through an asterisk (`*`)

If you use an `*` for the precision specifier, the next argument in the call (treated as an **int**) specifies the precision.

If you use asterisks for the width or the precision, or for both, the width argument must immediately follow the specifiers, followed by the precision argument, then the argument for the data to be converted.

`[.prec]`	**How Output Precision Is Affected**
(none)	Precision set to default: = 1 for *d,i,o,u,x,X* types = 6 for *e,E,f* types = All significant digits for *g,G* types = Print to first null character for *s* types = No effect on *c* types
`.0`	For *d,i,o,u,x* types, precision set to defaul for *e,E,f* types, no decimal point is printed.
`.n`	*n* characters or *n* decimal places are printed. If the output value has more than *n* characters, the output might be truncated or rounded. (Whether this happens depends on the type character.)
`.`	The argument list supplies the precision specifier, which must precede the actual argument being formatted.

No numeric characters will be output for a field (i.e., the field will be blank) if the following conditions are all met:

- you specify an explicit precision of 0
- the format specifier for the field is one of the integer formats (`d, i, o, u,` or `x`)
- the value to be printed is 0

How [.prec] Affects Conversion

Char Type	Effect of `[.prec]` (.n) on Conversion
`d`	Specifies that at least *n* digits are printed.
`i`	If input argument has less than *n* digits,
`o`	output value is left-padded x with zeros.
`u`	If input argument has more than *n* digits,
`x`	the output value is not truncated.
`X`	
`e`	Specifies that *n* characters are
`E`	printed after the decimal point, and

`f`	the last digit printed is rounded.
`g` `G`	Specifies that at most *n* significant digits are printed.
`c`	Has no effect on the output.
`s`	Specifies that no more than *n* characters are printed.

printf Conversion-Type Characters

See also

The information in this table is based on the assumption that no flag characters, width specifiers, precision specifiers, or input-size modifiers were included in the format specifier.

Note: Certain conventions accompany some of these format specifiers.

Type Char	Expected Input	Format of output
Numerics		
d	Integer	**signed** decimal **integer**
i	Integer	**signed** decimal **integer**
o	Integer	**unsigned** octal **integer**
u	Integer	**unsigned** decimal **integer**
x	Integer	**unsigned** hexadecimal **int** (with **a, b, c, d, e, f**)
X	Integer	**unsigned** hexadecimal **int** (with **A, B, C, D, E, F**)
f	Floating point	**signed** value of the form `[-]dddd.dddd`.
e	Floating point	**signed** value of the form `[-]d.dddd or e[+/-]ddd`
g	Floating point	**signed** value in either **e** or **f** form, based on given value and precision. Trailing zeros and the decimal point are printed if necessary.
E	Floating point	Same as **e**; with **E** for exponent.
G	Floating point	Same as **g;** with **E** for exponent if **e** format used
Characters		
c	Character	Single character
s	String pointer	Prints characters until a null-terminator is pressed or precision is reached
%	None	Prints the % character
Pointers		
n	Pointer to **int**	Stores (in the location pointed to by the input argument) a count of the chars written so far.
p	Pointer	Prints the input argument as a pointer; format depends on which memory model was used. It will be either XXXX:YYYY or YYYY (offset only).

Infinite floating-point numbers are printed as `+INF` and `-INF`.

An IEEE Not-A-Number is printed as `+NAN` or `-NAN`.

printf Input-size Modifiers

See also

These modifiers determine how printf functions interpret the next input argument, `arg[f]`.

Modifier	Type of arg	arg is interpreted as ...
F	Pointer (`p, s,`	A **far** pointer
N	and `n`)	A **near** pointer (**Note:** *N* can't be used with any conversion in **huge** model.)
h	*d i o u x X*	A **short int**
l	*d i o u x X*	A **long int**
	e E f g G	A **double**
L	*e E f g G*	A **long double**

These modifiers affect how all the printf functions interpret the data type of the corresponding input argument arg.

Both *F* and *N* reinterpret the input variable *arg*. Normally, the *arg* for a *%p*, *%s*, or *%n* conversion is a pointer of the default size for the memory model.

h, *l*, and *L* override the default size of the numeric data input arguments. Neither *h* nor *l* affects character (*c,s*) or pointer (*p,n*) types.

printf Format Specifier Conventions

See also

Certain conventions accompany some of the printf format specifiers for the following conversions:

- %e or %E
- %f
- %g or %G
- %x or %X

Note: Infinite floating point numbers are printed as +INF and -INF. An IEEE Not-a-Number is printed as +NAN or -NAN.

%e or %E Conversions

See also

The argument is converted to match the style

`[-] d.ddd...e[+/-]ddd`

where:

- one digit precedes the decimal point
- the number of digits after the decimal point is equal to the precision.
- the exponent always contains at least two digits

%f Conversions

See also

The argument is converted to decimal notation in the style

```
[-] ddd.ddd...
```

where the number of digits after the decimal point is equal to the precision (if a non-zero precision was given).

%g or %G Conversions

See also

The argument is printed in style **e, E** or **f,** with the precision specifying the number of significant digits.

Trailing zeros are removed from the result, and a decimal point appears only if necessary.

The argument is printed in style **e** or **f** (with some restraints) if **g** is the conversion character. Style **e** is used only if the exponent that results from the conversion is either greater than the precision or less than -4.

The argument is printed in style **E** if **G** is the conversion character.

%x or %X Conversions

See also

For **x** conversions, the letters **a, b, c, d, e**, and **f** appear in the output.

For **X** conversions, the letters **A, B, C, D, E**, and **F** appear in the output.

...printf functions

The ...printf functions include

fprintf	sends formatted output to a stream
printf	sends formatted output to stdout
sprintf	sends formatted output to a string
vfprintf	sends formatted output to a stream, using an argument list
vprintf	sends formatted output to *stdout*, using an argument list
vsprintf	sends formatted output to a string, using an argument list

putc, putwc

See also Portability

Syntax

```
#include <stdio.h>
int putc(int c, FILE *stream);
wint_t putwc(wint_t c, FILE *stream);
```

Description

Outputs a character to a stream.

putc is a macro that outputs the character *c* to the stream given by *stream*.

Return Value

On success, *putc* returns the character printed, *c.*

On error, *putc* returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

putch

See also Example Portability

Syntax

```
#include <conio.h>
int putch(int c);
```

Description

Outputs character to screen.

putch outputs the character *c* to the current text window. It is a text mode function performing direct video output to the console. *putch* does not translate linefeed characters (\n) into carriage-return/linefeed pairs.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *_directvideo*.

Note: This function should not be used in Win32s or Win32 GUI applications.

Return Value

On success, *putch* returns the character printed, *c*. On error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

putchar, putwchar

See also Example Portability

Syntax

```
#include <stdio.h>
int putchar(int c);
wint_t putwchar(wint_t c);
```

Description

putchar(c) is a macro defined to be *putc*(c, stdout).

Note: For Win32s or Win32 GUI applications, *stdout* must be redirected.

Return Value

On success, *putchar* returns the character *c*. On error, *putchar* returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

putenv, _wputenv

See also Example Portability

Syntax

```
#include <stdlib.h>
int putenv(const char *name);
int _wputenv(const wchar_t *name);
```

Description

Adds string to current environment.

putenv accepts the string *name* and adds it to the environment of the current process. For example,

```
putenv("PATH=C:\\BC");
```

putenv can also be used to modify an existing *name*. On DOS and OS/2, *name* must be uppercase. On other systems, *name* can be either uppercase or lowercase. *name* must not include the equal sign (**=**). You can set a variable to an empty value by specifying an empty string on the right side of the '**=**' sign.

putenv can be used only to modify the current program's environment. Once the program ends, the old environment is restored. The environment of the current process is passed to child processes, including any changes made by *putenv*.

Note that the string given to *putenv* must be static or global. Unpredictable results will occur if a local or dynamic string given to *putenv* is used after the string memory is released.

Return Value

On success, *putenv* returns 0; on failure, -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

puts, _putws

See also Example Portability

Syntax

```
#include <stdio.h>
int puts(const char *s);
int _putws(const wchar_t *s);
```

Description

Outputs a string to stdout.

puts copies the null-terminated string *s* to the standard output stream *stdout* and appends a newline character.

Note: For Win32s or Win32 GUI applications, *stdout* must be redirected.

Return Value

On successful completion, *puts* returns a nonnegative value. Otherwise, it returns a value of EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+		+

puttext

See also Example Portability

Syntax

```
#include <conio.h>
int puttext(int left, int top, int right, int bottom, void *source);
```

Description

Copies text from memory to the text mode screen.

puttext writes the contents of the memory area pointed to by *source* out to the onscreen rectangle defined by *left, top, right*, and *bottom*.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

puttext places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as

bytes = (*h* rows) x (*w* columns) x 2

puttext is a text mode function performing direct video output.

Note: This function should not be used in Win32s or Win32 GUI applications.

Return Value

puttext returns a nonzero value if the operation succeeds; it returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

putw

See also Example Portability

Syntax

```
#include <stdio.h>
int putw(int w, FILE *stream);
```

Description

Puts an integer on a stream.

putw outputs the integer *w* to the given stream. *putw* neither expects nor causes special alignment in the file.

Return Value

On success, *putw* returns the integer *w*. On error, *putw* returns EOF. Because EOF is a legitimate integer, use *ferror* to detect errors with *putw*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

qsort

See also Example Portability

Syntax

```
#include <stdlib.h>
void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)
  (const void *, const void *));
```

Description

Sorts using the quicksort algorithm.

qsort is an implementation of the "median of three" variant of the quicksort algorithm. *qsort* sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by *fcmp*.

- *base* points to the base (0th element) of the table to be sorted.
- *nelem* is the number of entries in the table.
- *width* is the size of each entry in the table, in bytes.

fcmp, the comparison function, must be used with the _USERENTRY calling convention.

fcmp accepts two arguments, *elem1* and *elem2*, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (**elem1* and **elem2*), and returns an integer based on the result of the comparison.

- **elem1* < **elem2* *fcmp* returns an integer < 0
- **elem1* == **elem2* *fcmp* returns 0
- **elem1* > **elem2* *fcmp* returns an integer > 0

In the comparison, the less-than symbol (<) means the left element should appear before the right element in the final, sorted sequence. Similarly, the greater-than (>) symbol means the left element should appear after the right element in the final, sorted sequence.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

raise

See also Example Portability

Syntax

```
#include <signal.h>
int raise(int sig);
```

Description

Sends a software signal to the executing program.

raise sends a signal of type *sig* to the program. If the program has installed a signal handler for the signal type specified by *sig*, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken.

The signal types currently defined in signal.h are noted here:

Signal	Description
SIGABRT	Abnormal termination
SIGFPE	Bad floating-point operation
SIGILL	Illegal instruction
SIGINT	Ctrl-C interrupt
SIGSEGV	Invalid access to storage
SIGTERM	Request for program termination
SIGUSR1	User-defined signal
SIGUSR2	User-defined signal
SIGUSR3	User-defined signal
SIGBREAK	Ctrl-Break interrupt

Note: SIGABRT isn't generated by Borland C++ during normal operation. It can, however, be generated by abort, raise, or unhandled exceptions.

Return Value

On succes, *raise* returns 0.

On error it returns nonzero.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

rand

See also Example Portability

Syntax

```
#include <stdlib.h>
int rand(void);
```

Description

Random number generator.

rand uses a multiplicative congruential random number generator with period 2 to the 32nd power to return successive pseudorandom numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h.

Return Value

rand returns the generated pseudorandom number.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

random

See also Example Portability

Syntax

```
#include <stdlib.h>
int random(int num);
```

Description

Random number generator.

random returns a random number between 0 and (*num*-1). *random(num)* is a macro defined in stdlib.h. Both *num* and the random number returned are integers.

Return Value

random returns a number between 0 and (*num*-1).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

randomize

See also Example Portability

Syntax

```
#include <stdlib.h>
#include <time.h>
void randomize(void);
```

Description

Initializes random number generator.

randomize initializes the random number generator with a random value.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

read

See also Example Portability

Syntax

```
#include <io.h>
int read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

read attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*.

For a file opened in text mode, read removes carriage returns and reports end-of-file when it reaches a Ctrl-Z.

The file handle *handle* is obtained from a *creat, open, dup*, or *dup2* call.

On disk files, *read* begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that *read* can read is UINT_MAX -1, because UINT_MAX is the same as -1, the error return indicator. UINT_MAX is defined in limits.h.

Return Value

On successful completion, *read* returns an integer indicating the number of bytes placed in the buffer. If the file was opened in text mode, *read* does not count carriage returns or Ctrl-Z characters in the number of bytes read.

On end-of-file, *read* returns 0. On error, *read* returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

readdir, wreaddir

See also Example Portability

Syntax

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
struct wdirent *wreaddir(wDIR *dirp)
```

Description

Reads the current entry from a directory stream.

readdir is available on POSIX-compliant UNIX systems.

The *readdir* function reads the current directory entry in the directory stream pointed to by *dirp*. The directory stream is advanced to the next entry.

The *readdir* function returns a pointer to a **dirent** structure that is overwritten by each call to the function on the same directory stream. The structure is not overwritten by a *readdir* call on a different directory stream.

The **dirent** structure corresponds to a single directory entry. It is defined in dirent.h and contains (in addition to other non-accessible members) the following member:

```
char  d_name[];
```

where *d_name* is an array of characters containing the null-terminated file name for the current directory entry. The size of the array is indeterminate; use *strlen* to determine the length of the file name.

All valid directory entries are returned, including subdirectories, "." and ".." entries, system files, hidden files, and volume labels. Unused or deleted directory entries are skipped.

A directory entry can be created or deleted while a directory stream is being read, but *readdir* might or might not return the affected directory entry. Rewinding the directory with *rewinddir* or reopening it with *opendir* ensures that *readdir* will reflect the current state of the directory.

The *wreaddir* function is the Unicode version of *readdir*. It uses the **wdirent** structure but otherwise is similar to *readdir*.

Return Value

On success, *readdir* returns a pointer to the current directory entry for the directory stream.

If the end of the directory has been reached, or *dirp* does not refer to an open directory stream, *readdir* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

realloc

See also Example Portability

Syntax

```
#include <stdlib.h>
void *realloc(void *block, size_t size);
```

Description

Reallocates main memory.

realloc attempts to shrink or expand the previously allocated block to *size* bytes. If *size* is zero, the memory block is freed and NULL is returned. The *block* argument points to a memory block previously obtained by calling *malloc, calloc*, or *realloc*. If *block* is a NULL pointer, *realloc* works just like *malloc*.

realloc adjusts the size of the allocated block to *size*, copying the contents to a new location if necessary.

Return Value

realloc returns the address of the reallocated block, which can be different than the address of the original block.

If the block cannot be reallocated, *realloc* returns NULL.

If the value of *size* is 0, the memory block is freed and *realloc* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

remove, _wremove

See also Example Portability

Syntax

```
#include <stdio.h>
int remove(const char *filename);
int _wremove(const wchar_t *filename);
```

Description

Removes a file.

remove deletes the file specified by *filename*. It is a macro that simply translates its call to a call to *unlink*. If your file is open, be sure to close it before removing it.

The *filename* string can include a full path.

Return Value

On successful completion, *remove* returns 0. On error, it returns -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

rename, _wrename

Example Portability

Syntax

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
int _wrename(const wchar_t *oldname, const wchar_t *newname);
```

Description

Renames a file.

rename changes the name of a file from *oldname* to *newname*. If a drive specifier is given in *newname*, the specifier must be the same as that given in *oldname*.

Directories in *oldname* and *newname* need not be the same, so *rename* can be used to move a file from one directory to another. Wildcards are not allowed.

This function will fail (EEXIST) if either file is currently open in any process.

Return Value

On success, *rename* returns 0.

On error (if the file cannot be renamed), it returns -1 and the global variable errno is set to one of the following values:

EEXIST	Permission denied: file already exists.
ENOENT	No such file or directory
ENOTSAM	Not same device

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

rewind

See also Example Portability

Syntax

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

Repositions a file pointer to the beginning of a stream.

rewind(stream) is equivalent to fseek(stream, 0L, SEEK_SET), except that *rewind* clears the end-of-file and error indicators, while *fseek* clears the end-of-file indicator only.

After *rewind*, the next operation on an update file can be either input or output.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

rewinddir, wrewinddir

See also Example Portability

Syntax

```
#include <dirent.h>
void rewinddir(DIR *dirp);
void wrewinddir(wDIR *dirp);
```

Description

Resets a directory stream to the first entry.

rewinddir is available on POSIX-compliant UNIX systems.

The *rewinddir* function repositions the directory stream *dirp* at the first entry in the directory. It also ensures that the directory stream accurately reflects any directory entries that might have been created or deleted since the last *opendir* or *rewinddir* on that directory stream.

wrewinddir is the Unicode version of *rewinddir*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_rmdir, _wrmdir

See also Example Portability

Syntax

```
#include <dir.h>
int _rmdir(const char *path);
int _wrmdir(const wchar_t *path);
```

Description

Removes a directory.

_rmdir deletes the directory whose path is given by *path*. The directory named by *path*

- must be empty
- must not be the current working directory
- must not be the root directory

Return Value

_rmdir returns 0 if the directory is successfully deleted. A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file function not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

rmtmp

See also Example Portability

Syntax

```
#include <stdio.h>
int rmtmp(void);
```

Description

Removes temporary files.

The *rmtmp* function closes and deletes all open temporary file streams, which were previously created with *tmpfile*. The current directory must the same as when the files were created, or the files will not be deleted.

Return Value

rmtmp returns the total number of temporary files it closed and deleted.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_rotl, _rotr

See also Example Portability

Syntax

```
#include <stdlib.h>
unsigned short _rotl(unsigned short value, int count);
unsigned short _rotr(unsigned short value, int count);
```

Description

Bit-rotates an **unsigned** short integer value to the left or right.

_rotl rotates the given *value* to the left *count* bits.

_rotr rotates the given *value* to the right *count* bits.

Return Value

_rotl, and *_rotr* return the rotated integer:

- *_rotl* returns the value of *value* left-rotated *count* bits.
- *_rotr* returns the value of *value* right-rotated *count* bits.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_rtl_chmod, _wrtl_chmod

See also Example Portability

Syntax

```
#include <io.h>
int _rtl_chmod(const char *path, int func [, int attrib]);
int _wrtl_chmod(const wchar_t *path, int func, ... );
```

Description

Gets or sets file attributes.

Note: The *_rtl_chmod* function replaces *_chmod* which is obsolete

_rtl_chmod can either fetch or set file attributes. If *func* is 0, *_rtl_chmod* returns the current attributes for the file. If *func* is 1, the attribute is set to *attrib*.

attrib can be one of the following symbolic constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

Return Value

On success, *_rtl_chmod* returns the file attribute word.

On error, it returns a value of -1 and sets the global variable errno to one of the following values:

ENOENT	Path or filename not found
EACCES	Permission denied

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

_rtl_close

See also Example Portability

Syntax

```
#include <io.h>
int _rtl_close(int handle);
```

Description

Closes a file.

Note: This function replaces *_close* which is obsolete

The *_rtl_close* function closes the file associated with *handle*, a file handle obtained from a call to *creat, creatnew, creattemp, dup, dup2, open, _rtl_creat*, or *_rtl_open*.

It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one.

Return Value

On success, *_rtl_close* returns 0.

On error (if it fails because *handle* is not the handle of a valid, open file), *_rtl_close* returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

_rtl_creat, _wrtl_creat

See also Example Portability

Syntax

```
#include <io.h>
int _rtl_creat(const char *path, int attrib);
int _wrtl_creat(const wchar_t *path, int attrib);
```

Description

Creates a new file or overwrites an existing one.

Note: The *_rtl_creat* function replaces *_creat* which is obsolete

_rtl_creat opens the file specified by *path*. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The *attrib* argument is an ORed combination of one or more of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return Value

On success, *_rtl_creat* returns the new file handle (a non-negative integer).

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

_rtl_heapwalk

See also Example Portability

Syntax

```
#include <malloc.h>
int _rtl_heapwalk(_HEAPINFO *hi);
```

Description

Inspects the heap node by node.

Note: This function replaces *_heapwalk* which is obsolete.

_rtl_heapwalk assumes the heap is correct. Use _heapchk to verify the heap before using *_rtl_heapwalk.* _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to *_rtl_heapwalk*.

_rtl_heapwalk receives a pointer to a structure of type _HEAPINFO (declared in malloc.h).

For the first call to *_rtl_heapwalk*, set the *hi._pentry* field to NULL. *_rtl_heapwalk* returns with *hi._pentry* containing the address of the first block.

hi._size	holds the size of the block in bytes.
hi._useflag	is a flag that is set to _USEDENTRY if the block is currently in use. If the block is free, *hi._useflag* is set to _FREEENTRY.

Return Value

This function returns one of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPBADPTR	The _pentry field does not point to a valid heap block
_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The _heapinfo block contains valid information about the next heap block

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

_rtl_open, _wrtl_open

See also Example Portability

Syntax

```
#include <io.h>
int _rtl_open(const char *filename, int oflags);
int _wrtl_open(const wchar_t *path, int oflags);
```

Description

Opens a file for reading or writing.

Note: The *_rtl_open* function replaces *_open* which is obsolete.

_rtl_open opens the file specified by *filename*, then prepares it for reading or writing, as determined by the value of *oflags*. The file is always opened in binary mode.

oflags uses the flags from the following two lists. Only one flag from List 1 can be used (and one *must* be used) and the flags in List 2 can be used in any logical combination.

List 1: Read/write flags

O_RDONLY	Open for reading.
O_WRONLY	Open for writing.
O_RDWR	Open for reading and writing.

The following additional values can be included in *oflags* (using an OR operation):

List 2: Other access flags

O_NOINHERIT	The file is not passed to child programs.
SH_COMPAT	Allow other opens with SH_COMPAT. All other openings of a file with the SH_COMPAT flag must be opened using SH_COMPAT flag. You can request a file open that uses SH_COMPAT logically OR'ed with some other flag (for example, SH_COMPAT | SH_DENWR is allowed). The call will fail if the file has already been opened in any other shared mode.
SH_DENYRW	Only the current handle can have access to the file.
SH_DENWR	Allow only reads from any other open to the file.
SH_DENYRD	Allow only writes from any other open to the file.
SH_DENYNO	Allow other shared opens to the file, but not other SH_COMPAT opens.

Note: These symbolic constants are defined in fcntl.h and share.h.

Only one of the SH_DENY*xx* values can be included in a single *_rtl_open* routine. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

Return Value

On success:*_rtl_open* returns a non-negative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	Path or file not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

_rtl_read

See also Example Portability

Syntax

```
#include <dos.h>
int _rtl_read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

Note: This function replaces *_read* which is obsolete.

This function reads *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*. When a file is opened in text mode, *_rtl_read* does not remove carriage returns.

The argument *handle* is a file handle obtained from a creat, open, dup, or dup2 call.

On disk files, *_rtl_read* begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes it can read is UINT_MAX -1 (because UINT_MAX is the same as -1, the error return indicator). UINT_MAX is defined in limits.h.

Return Value

On success, *_rtl_read* returns either

- a positive integer, indicating the number of bytes placed in the buffer
- zero, indicating end-of-file

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

_rtl_write

See also Example Portability

Syntax

```
#include <io.h>
int _rtl_write(int handle void *buf unsigned len);
```

Description

Writes to a file.

Note: This function replaces *_write* which is obsolete.

_rtl_write attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle*.

The maximum number of bytes that *_rtl_write* can write is UINT_MAX -1 (because UINT_MAX is the same as -1), which is the error return indicator for *_rtl_write*. UINT_MAX is defined in limits.h. *_rtl_write* does not translate a linefeed character (LF) to a CR/LF pair because all its files are binary files.

If the number of bytes actually written is less than that requested the condition should be considered an error and probably indicates a full disk.

For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

For files opened with the O_APPEND option, the file pointer is not positioned to EOF before writing the data.

Return Value

On success, *_rtl_write* returns number of bytes written.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			

scanf, wscanf

See also Example Portability

Syntax

```
#include <stdio.h>
int scanf(const char *format[, address, ...]);
int wscanf(const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from the stdin stream.

Note: For Win32s or Win32 GUI applications, stdin must be redirected.

The *scanf* function:

- scans a series of input fields one character at a time
- formats each field according to a corresponding format specifier passed in the format string **format*.
- *vsscanf* scans and formats input from a string, using an argument list

There must be one format specifier and address for each input field.

scanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely. For details about why this might happen, see When ...scanf Stops Scanning.

Warning: *scanf* often leads to unexpected results if you diverge from an expected pattern. You must provide information that tells scanf how to synchronize at the end of a line.

The combination of gets or fgets followed by *sscanf* is safe and easy, and therefore recommended over *scanf*.

Return Value

On success, *scanf* returns the number of input fields successfully scanned, converted, and stored. The return value does not include scanned fields that were not stored.

On error:

- if no fields were stored, *scanf* returns 0.
- if *scanf* attempts to read at end-of-file or at end-of-string, it returns EOF.

More About scanf

Unicode input format specifiers

Argument-type Modifiers

Assignment Suppression

Format Specifiers

Format Specifier Conventions

Format String

Input Fields

Pointer-size Modifiers

Type Characters

Width Specifiers

When ...scanf Functions Stop Scanning

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

The scanf Format String

See also

The format string controls how each ...*scanf* function scans, converts, and stores its input fields.

The format string is a character string that contains three types of objects:

- *whitespace characters*
- *non-whitespace characters*
- *format specifiers*

Whitespace Characters

The whitespace characters are blank, tab (`\t`) or newline (`\n`).

If a ...*scanf* function encounters a whitespace character in the format string, it reads, but does not store, all consecutive whitespace characters up to the next non-whitespace character in the input.

Trailing whitespace is left unread (including a newline), unless explicitly matched in the format string.

Non-whitespace Characters

The non-whitespace characters are all other ASCII characters except the percent sign (%).

If a ...*scanf* function encounters a non-whitespace character in the format string, it will read, but not store, a matching non-whitespace character.

Format Specifiers

The format specifiers direct the ...*scanf* functions to read and convert characters from the input field into specific types of values, then store them in the locations given by the address arguments.

Warning: Each format specifier must have an address argument. If there are more format specs than addresses, the results are unpredictable and likely disastrous.

Excess address arguments (more than required by the format) are ignored.

scanf Format Specifiers

See also

In ...scanf format strings, format specifiers have the following form:

```
% [*] [width] [F|N] [h|l|L] type_char
```

Each format specifier begins with the percent character (`%`).

After the `%` come the following, in this order:

Component	Optional/Required	What It Is/Does
`[*]`	(Optional)	Assignment-suppression character. Suppresses assignment of the next input field.
`[width]`	(Optional)	Width specifier. pecifies maximum number of characters to read; fewer characters might be read if the ...*scanf* function encounters a whitespace or unconvertible character.
`[F|N]`	(Optional)	Pointer size modifier. Overrides default size of address argument: *N* = **near** pointer *F* = **far** pointer
`[h|l|L]`	(Optional)	Argument-type modifier. Overrides default type of address argument: *h* = **short int** *l* = **long int**, if `type_char` specifies integer conversion *l* = **double**, if `type_char` specifies floating-point conversion *L* = **long double**, (valid only with floating-point conversion)
`type_char`	(Required)	Type character

Type Characters

See also

The information in this table is based on the assumption that no optional characters, specifiers, or modifiers (*, width, or size) were included in the format specifier.

Note: Certain conventions accompany some of these format specifiers.

Type	Expected input	Type of argument
Numerics		
d	Decimal integer	Pointer to **int** (**int** *arg)
D	Decimal integer	Pointer to **long** (**long** *arg)
e,E	Floating point	Pointer to **float** (**float** *arg)
f	Floating point	Pointer to **float** (**float** *arg)
g,G	Floating point	Pointer to **float** (**float** *arg)
o	Octal integer	Pointer to **int** (**int** *arg)
O	Octal integer	Pointer to **long** (**long** *arg)
i	Decimal, octal, or hexadecimal integer	Pointer to **int** (**int** *arg)
I	Decimal, octal, or hexadecimal integer	Pointer to **long** (**long** *arg)
u	Unsigned decimal integer	Pointer to **unsigned int** (**unsigned int** *arg)
U	Unsigned decimal integer	Pointer to **unsigned long** (**unsigned long** *arg)
x	Hexadecimal integer	Pointer to **int** (**int** *arg)
X	Hexadecimal integer	Pointer to **int** (**int** *arg)
Characters		
s	Character string	Pointer to array of **chars** (**char** *arg[])
c	Character	Pointer to **char** (**char** *arg) if a field width is given along with the *c*-type character (such as %5c)
	Pointer to array of *W* chars (**char** *arg[W])	
%	% character	No conversion done; the % is stored
Pointers		
n	Pointer to **int** (**int** *arg). The number of characters read successfully up to %n is stored in this **int**.	
p	Hexadecimal form	Pointer to an object (**far*** or **near***)
	YYYY:ZZZZ or ZZZZ	%p conversions default to the pointer size native to the memory model

Input Fields for Scanf Functions

See also

In a ...*scanf* function, any one of the following is an input field:

- all characters up to (but not including) the next whitespace character
- all characters up to the first one that can't be converted under the current format specifier (such as an 8 or 9 under octal format)
- up to *n* characters, where *n* is the specified field width

Assignment-suppression Character

See also

The assignment-suppression character is an asterisk (`*`), not to be confused with the C indirection (pointer) operator.

If the asterisk follows the percent sign (`%`) in a format specifier, the next input field will be scanned but it won't be assigned to the next address argument.

The suppressed input data is assumed to be of the type specified by the type character that follows the asterisk character.

Width Specifiers

See also

The width specifier (*n*), a decimal integer, controls the maximum number of characters to be read from the current input field.

Up to *n* characters are read, converted, and stored in the current address argument.

If the input field contains fewer than *n* characters, the ...*scanf* function reads all the characters in the field, then proceeds with the next field and format specifier.

The success of literal matches and suppressed assignments is not directly determinable.

If the ...*scanf* function encounters a whitespace or non-convertible character before it reads "width" characters, it:

- reads, converts, and stores the characters read so far, then
- attends to the next format specifier.

A non-convertible character is one that can't be converted according to the given format (8 or 9 when the format is octal, *J* or *K* when the format is hexadecimal or decimal, etc.).

Pointer-size and Argument-type Modifiers

See also

These modifiers affect how ...*scanf* functions interpret the corresponding address argument `arg[f]`.

Pointer-size Modifiers

Pointer-size modifiers override the default or declared size of *arg*.

Modifier	arg Interpreted As...
F	**Far** pointer
N	**Near** pointer (Can't be used with any conversion in huge model)

Argument-type Modifiers

Argument-type modifiers indicate which type of the following input data is to be used (h = **short**, l = **long**, L = **long double**).

The input data is converted to the specified version, and the *arg* for that input data should point to an object of corresponding size.

Modifier	For This Type	Convert Input to...
h	`d i o u x`	**short int**; store in **short** object
	`D I O U X`	(No effect)
	`e f c s n p`	(No effect)
l	`d i o u x`	**long int**; store in **long** object
	`e f g`	**double**; store in **double** object
	`D I O U X`	(No effect)
	`c s n p`	(No effect)
L	`e f g`	**long double**; store in **long double** object
	(all others)	(No effect)

Format Specifier Conventions

See also

Certain conventions accompany some of the ...scanf format specifiers for the following conversions:

single character (%c)

character array (%[W]c)

string (%s)

floating-point (%e, %E, %f, %g, and %G)

unsigned (%d, %i, %o, %x, %D, %I, %O, %X, %c, %n)

search sets(%[...], %[^...])

Single Character Conversion (%c)

See also

This specification reads the next character, including a whitespace character.

To skip one whitespace character and read the next non-whitespace character, use `%1s`.

Character Array Conversion (%[W]c)

`[W]` = width specification

The address argument is a pointer to an array of characters (**`char`** *`arg[W]`*).

The array consists of *W* elements.

String Conversion (%s)

See also

The address argument is a pointer to an array of characters (**`char`** *`arg[]`*).

The array size must be *at least* (*n*+1) bytes, where *n* = the length of string *s* (in characters).

A space or newline character terminates the input field.

A null terminator is automatically appended to the string and stored as the last element in the array.

Floating-point Conversions (%e, %E, %f, %g, and %G)

See also

Floating-point numbers in the input field must conform to the following generic format:

`[+/-] ddddddddd [.] dddd [E|e] [+/-] ddd`

where `[item]` indicates that *item* is optional, and `ddd` represents digits (decimal, octal, or hexadecimal).

In addition, +INF, -INF, +NAN, and -NAN are recognized as floating-point numbers. The sign (+ or -) and capitalization are required.

Unsigned Conversions (%d, %i, %o, %x, %D, %I, %O, %X, %c, and %n)

See also

A pointer to **unsigned** character, **unsigned** integer, or **unsigned long** can be used in any conversion where a pointer to a character, integer, or **long** is allowed.

Search Set Conversion (%[search_set])

See also Examples

The set of characters surrounded by brackets can be substituted for the *s*-type character.

The address argument is a pointer to an array of characters (**`char`** *`arg[]`*).

These brackets surround a set of characters that define a *search set* of possible characters making up the string (the input field).

If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the brackets.

(Normally, a caret will be included in the inverted search set unless explicitly listed somewhere after the first caret.)

The input field is a string not delimited by whitespace. *...scanf* reads the corresponding input field up to the first character it reaches that does not appear in the search set (or in the inverted search set).

Rules covering search set ranges

1. The character prior to the hyphen (-) must be lexically less than the one after it.
2. The hyphen must not be the first or last character in the set. (If it is first or last, it is considered to just be the hyphen character, not a range definer.)
3. The characters on either side of the hyphen must be the ends of the range and not part of some other range.

Examples

`%[abcd]`	Searches the input field for any of the characters *a, b, c*, and *d*
`%[^abcd]`	Searches the input field for any characters except *a, b, c*, and *d*

You can also use a range facility shortcut `[<first>-<last>]` to define a range of letters or numerals in the search set.

Examples

To catch all decimal digits, you could define the search set with the explicit search set: `%[0123456789]` or with the range shortcut: `%[0-9]`

To catch alphanumeric characters, you could use the following shortcuts:

`%[A-Z]`	Catches all uppercase letters
`%[0-9A-Za-z]`	Catches all decimal digits and all letters
`%[A-FT-Z]`	Catches all uppercase letters from *A* through *F* and from *T* through *Z*.

When ...scanf Functions Stop Scanning

See also

A ...*scanf* function might stop scanning a particular input field before reaching the normal field-end character (whitespace), or it might terminate entirely.

Stop and Skip to Next Input Field

...*scanf* functions stop scanning and storing the current input field and proceed to the next one if any of the following occurs:

- An assignment-suppression character (*) appears after the % in the format specifier. The current input field is scanned but not stored.
- width characters have been read.
- The next character read can't be converted under the current format (for example, an *A* when the format is decimal).
- The next character in the input field does not appear in the search set (or does appear in an inverted search set).

When *scanf* stops scanning the current input field for one of these reasons, it assumes that the next character is unread and is either

- the first character of the following input field, or
- the first character in a subsequent read operation on the input.

Terminate

...*scanf* functions will terminate under the following circumstances:

1. The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
2. The next character in the input field is EOF.
3. The format string has been exhausted.

If a character sequence that is not part of a format specifier occurs in the format string, it must match the current sequence of characters in the input field.

...*scanf* functions will scan but not store the matched characters.

When a conflicting character occurs, it remains in the input field as if the *...scanf* function never read it.

...scanf functions

The ..scanf functions include

fscanf	scans and formats input from a stream
scanf	scans and formats input from stdin
sscanf	scans and formats input from a string
vfscanf	scans and formats input from a stream, using an argument list
vscanf	scans and formats input from stdin using an argument list
vsscanf	scans and formats input from a string, using an argument list

_searchenv, _wsearchenv

See also Example Portability

Syntax

```
#include <stdlib.h>
void _searchenv(const char *file, const char *varname, char *buf);
void _wsearchenv(const wchar_t *file, const wchar_t *varname, wchar_t *buf);
```

Description

Searches an environment path for a file.

_searchenv attempts to locate *file*, searching along the path specified by the operating system environment variable *varname*. Typical environment variables that contain paths are PATH, LIB, and INCLUDE.

_searchenv searches for the file in the current directory of the current drive first. If the file is not found there, the environment variable *varname* is fetched, and each directory in the path it specifies is searched in turn until the file is found, or the path is exhausted.

When the file is located, the full path name is stored in the buffer pointed to by *buf*. This string can be used in a call to access the file (for example, with *fopen* or *exec...*). The buffer is assumed to be large enough to store any possible file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at *buf*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

searchpath, wsearchpath

See also Example Portability

Syntax

```
#include <dir.h>
char *searchpath(const char *file);
wchar_t *wsearchpath( const wchar_t *file );
```

Description

Searches the operating system path for a file.

searchpath attempts to locate *file*, searching along the operating system path, which is the PATH=... string in the environment. A pointer to the complete path-name string is returned as the function value.

searchpath searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found, or the path is exhausted.

When the file is located, a string is returned containing the full path name. This string can be used in a call to access the file (for example, with *fopen* or *exec...*).

The string returned is located in a static buffer and is overwritten on each subsequent call to *searchpath*.

Return Value

searchpath returns a pointer to a file name string if the file is successfully located; otherwise, *searchpath* returns null.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_searchstr, _wsearchstr

See also Example Portability

Syntax

```
#include <stdlib.h>
void _searchstr(const char *file, const char *ipath, char *buf);
void _wsearchstr(const wchar_t *file, const wchar_t *ipath,wchar_t
  *pathname);
```

Description

Searches a list of directories for a file.

_*searchstr* attempts to locate *file*, searching along the path specified by the string *ipath*.

_*searchstr* searches for the file in the current directory of the current drive first. If the file is not found there, each directory in *ipath* is searched in turn until the file is found, or the path is exhausted. The directories in *ipath* must be separated by semicolons.

When the file is located, the full path name is stored in the buffer pointed by by *buf*. This string can be used in a call to access the file (for example, with *fopen* or *exec...*). The buffer is assumed to be large enough to store any possible file name. The constant _MAX_PATH defined in stdlib.h, is the size of the largest file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at *buf*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

segread

See also Example Portability

Syntax

```
#include <dos.h>
void segread(struct SREGS *segp);
```

Description

Reads segment registers.

segread places the current values of the segment registers into the structure pointed to by *segp*.

This call is intended for use with *intdosx* and *int86x*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

setbuf

See also Example Portability

Syntax

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Description

Assigns buffering to a stream.

setbuf causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after *stream* has been opened.

If *buf* is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h).

stdin and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered. *setbuf* can be used to change the buffering style used.

Unbuffered means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

setbuf produces unpredictable results unless it is called immediately after opening *stream* or after a call to *fseek*. Calling *setbuf* after *stream* has been unbuffered is legal and will not cause problems.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

setcbrk

See also Example Portability

Syntax

```
#include <dos.h>
int setcbrk(int cbrkvalue);
```

Description

Sets control-break setting.

setcbrk uses the DOS system call 0x33 to turn control-break checking on or off.

value = 0	Turns checking off (check only during I/O to console, printer, or communications devices).
value = 1	Turns checking on (check at every system call).

Return Value

setcbrk returns *cbrkvalue*, the value passed.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

_setcursortype

Example Portability

Syntax

```
#include <conio.h>
void _setcursortype(int cur_t);
```

Description

Selects cursor appearance.

Sets the cursor type to

_NOCURSOR	Turns off the cursor
_NORMALCURSOR	Normal underscore cursor
_SOLIDCURSOR	Solid block cursor

Note: Do not use this function for Win32s or Win32 GUI applications.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

setdta

See also Example Portability

Syntax

```
#include <dos.h>
void setdta(char far *dta);
```

Description

Sets disk-transfer address.

setdta changes the current setting of the DOS disk-transfer address (DTA) to the value given by *dta*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				

setjmp

See also Example Portability

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf jmpb);
```

Description

Sets up for nonlocal goto.

setjmp captures the complete *task state* in *jmpb* and returns 0.

A later call to *longjmp* with *jmpb* restores the captured task state and returns in such a way that *setjmp* appears to have returned with the value *val*.

A task state includes

Win 16	Win 32
All segment registers CS, DS, ES, SS	No segment registers are saved
Register variables	Register variables
DI and SI	EBX, EDI, ESI
Stack pointer SP	Stack pointer ESP
Frame pointer BP	Frame pointer EBP
Flags	Flags are not saved

A task state is complete enough that *setjmp* can be used to implement co-routines.

setjmp must be called before *longjmp*. The routine that calls *setjmp* and sets up *jmpb* must still be active and cannot have returned before the *longjmp* is called. If it has returned, the results are unpredictable.

setjmp is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

DOS Users

You cannot use *setjmp* and *longjmp* for implementing co-routines if your program is overlaid. Normally, *setjmp* and *longjmp* save and restore all the registers needed for co-routines, but the overlay manager needs to keep track of stack contents and assumes there is only one stack. When you implement co-routines there are usually either two stacks or two partitions of one stack, and the overlay manager will not track them properly.

You can have background tasks that run with their own stacks or sections of stack, but you must ensure that the background tasks do not invoke any overlaid code, and you must not use the overlay versions of *setjmp* or *longjmp* to switch to and from background. When you avoid using overlay code or support routines, the existence of the background stacks does not disturb the overlay manager.

Return Value

setjmp returns 0 when it is initially called. If the return is from a call to *longjmp*, *setjmp* returns a nonzero value (as in the example).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

setlocale, _wsetlocale

See also Example Portability

Syntax

```
#include <locale.h>
char *setlocale(int category, const char *locale);
wchar_t * _wsetlocale( int category, const wchar_t *locale);
```

Description

Use the *setlocale* to select or query a locale.

Borland C++ supports all locales supported in NT 3.5x and Win95/NT 4.0 operating systems. See your system documentation for details.

The possible values for the *category* argument are as follows:

Value	Affect
LC_ALL	Affects all the following categories
LC_COLLATE	Affects *strcoll* and *strxfrm*
LC_CTYPE	Affects single-byte character handling functions. The *mbstowcs* and *mbtowc* functions are not affected.
LC_MONETARY	Affects monetary formatting by the *localeconv* function
LC_NUMERIC	Affects the decimal point of non-monetary data formatting. This includes the *printf* family of functions, and the information returned by *localeconv*.
LC_TIME	Affects *strftime*

The *locale* argument is a pointer to the name of the locale or named locale category. Passing a NULL pointer returns the current locale in effect. Passing a pointer that points to a null string requests *setlocale* to look for environment variables to determine which locale to set. The locale names are **not** case sensitive.

When *setlocale* is unable to honor a locale request, the preexisting locale in effect is unchanged and a null pointer is returned.

If the *locale* argument is a NULL pointer, the locale string for the category is returned. If *category* is LC_ALL, a complete locale string is returned. The structure of the complete locale string consists of the names of all the categories in the current locale concatenated and separated by semicolons. This string can be used as the locale parameter when calling *setlocale* with any of the LC_*xxx* values. This will reinstate all the locale categories that are named in the complete locale string, and allows saving and restoring of locale states. If the complete locale string is used with a single category, for example, LC_TIME, only that category will be restored from the locale string.

If an empty string "" is used as the locale parameter an implementation-defined locale is used. This is the ANSI C specified behavior.

To take advantage of dynamically loadable locales in your application, define _ _USELOCALES_ _ for each module. If _ _USELOCALES_ _ is not defined, all locale-sensitive functions and macros will work only with the default C locale.

If a NULL pointer is used as the argument for the *locale* parameter, *setlocale* returns a string that specifies the current locale in effect. If the *category* parameter specifies a single category, such as LC_COLLATE, the string pointed to will be the name of that category. If LC_ALL is used as the *category* parameter then the string pointed to will be a full locale string that will indicate the name of each category in effect.

```
    .
    .
    .
localenameptr = setlocale( LC_COLLATE, NULL );
```

```
if (localenameptr)
   printf( "%s\n", localenameptr );
    .
    .
    .
```

The output here will be one of the module names together with the specified code page. For example, the output could be `LC_COLLATE = English_United States.437`.

```
    .
    .
    .
localenameptr = setlocale( LC_ALL, NULL );

if (localenameptr)
   printf( "%s\n", localenameptr );
    .
    .
    .
```

An example of the output here could be the following:

```
   LC_COLLATE=English_United States.437;
   LC_TIME=English_United States.437;
   LC_CTYPE=English_United States.437;
```

Each category in this full string is delimited by a semicolon. This string can be copied and saved by an application and then used again to restore the same locale categories at another time. Each delimited name corresponds to the locale category constants defined in locale.h. Therefore, the first name is the name of the LC_COLLATE category, the second is the LC_CTYPE category, and so on. Any other categories named in the locale.h header file are reserved for future implementation.

To set all default categories for the specified French locale:

```
   setlocale( LC_ALL, "French_France.850" );
```

To find out which code page is currently being used:

```
   localenameptr = setlocale( LC_ALL, NULL );
```

Return value

If selection is successful, *setlocale* returns a pointer to a string that is associated with the selected category (or possibly all categories) for the new locale.

If UNICODE is defined, *_wsetlocale* returns a **wchar_t** string.

On failure, a NULL pointer is returned and the locale is unchanged. All other possible returns are discussed in the Remarks section above.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

setmem

See also Example Portability

Syntax

```
#include <mem.h>
void setmem(void *dest, unsigned length, char value);
```

Description

Assigns a value to a range of memory.

setmem sets a block of *length* bytes, pointed to by *dest*, to the byte *value*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

setmode

See also Example Portability

Syntax

```
#include <io.h>
int setmode(int handle, int amode);
```

Description

Sets mode of an open file.

setmode sets the mode of the open file associated with *handle* to either binary or text. The argument *amode* must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.)

Return Value

setmode returns the previous translation mode if successful. On error it returns -1 and sets the global variable errno to

EINVAL	Invalid argument

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

setvbuf

See also Example Portability

Syntax

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

Assigns buffering to a stream.

setvbuf causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given stream is opened.

If *buf* is null, a buffer will be allocated using *malloc*; the buffer will use *size* as the amount allocated. The buffer will be automatically freed on close. The *size* parameter specifies the buffer size and must be greater than zero.

The parameter size is limited by the constant UINT_MAX as defined in limits.h.

stdin and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered. *Unbuffered* means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

The *type* parameter is one of the following:

_IOFBF	*fully buffered* file. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.
_IOLBF	*line buffered* file. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	*unbuffered* file. The *buf* and *size* parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

On success, *setvbuf* returns 0.

On error (if an invalid value is given for *type* or *size*, or if there is not enough space to allocate a buffer), it returns nonzero.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

setverify

See also Example Portability

Syntax

```
#include <dos.h>
void setverify(int value);
```

Description

Sets the state of the verify flag in the operating system.

setverify sets the current state of the verify flag to *value*, which can be either 0 (off) or 1 (on).

The verify flag controls output to the disk. When verify is off, writes are not verified; when verify is on, all disk writes are verified to ensure proper writing of the data.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+				+

signal

See also Example Portability

Syntax

```
#include <signal.h>
void (_USERENTRY *signal(int sig, void (_USERENTRY *func)
                                  (int sig[, int subcode])))(int);
```

Description

Specifies signal-handling actions.

signal determines how receipt of signal number *sig* will subsequently be treated. You can install a user-specified handler routine (specified by the argument *func*) or use one of the two predefined handlers, SIG_DFL and SIG_IGN, in signal.h. The function *func* must be used with the _USERENTRY calling convention.

A routine that catches a signal (such as a floating point) also clears the signal. To continue to receive signals, a signal handler must be reinstalled by calling signal again.

Function Pointer	Description
SIG_DFL	Terminates the program
SIG_ERR	Indicates an error return from signal
SIG_IGN	Ignore this type signal

The following table shows signal types and their defaults:

Signal Type	Description
SIGBREAK	Keyboard must be in raw mode.
SIGABRT	Abnormal termination. Default action is equivalent to calling *_exit*(3).
SIGFPE	Arithmetic error caused by division by 0, invalid operation, and the like. Default action is equivalent to calling *_exit*(1).
SIGILL	Illegal operation. Default action is equivalent to calling *_exit*(1).
SIGINT	Ctrl-C interrupt. Default action is to do an INT 23h.
SIGSEGV	Illegal storage access. Default action is equivalent to calling *_exit*(1).
SIGTERM	Request for program termination. Default action is equivalent to calling *_exit*(1).
SIGUSR1, SIGUSR2, SIGUSR3	User-defined signals (available only in Win32) can be generated only by calling *raise*. Default action is to ignore the signal

signal.h defines a type called *sig_atomic_t*, the largest integer type the processor can load or store atomically in the presence of asynchronous interrupts (for the 8086 family, this is a 16-bit word, for 80386 and higher number processors, it is a 32-bit word -- a Borland C++ integer).

When a signal is generated by the *raise* function or by an external event, the following two things happen:

- If a user-specified handler has been installed for the signal, the action for that signal type is set to SIG_DFL.
- The user-specified function is called with the signal type as the parameter.

User-specified handler functions can terminate by a return or by a call to *abort, _exit, exit*, or *longjmp*. If your handler function is expected to continue to receive and handle more signals, you must have the handler function call *signal* again.

Borland C++ implements an extension to ANSI C when the signal type is SIGFPE, SIGSEGV, or SIGILL. The user-specified handler function is called with one or two extra parameters. If SIGFPE, SIGSEGV, or SIGILL has been raised as the result of an explicit call to the *raise* function, the user-specified handler is

called with one extra parameter, an integer specifying that the handler is being explicitly invoked. The explicit activation values for SIGFPE, SIGSEGV and SIGILL are as follows

Note: Declarations of these types are defined in float.h.

SIGSEGV signal	Meaning
SIGFPE	FPE_EXPLICITGEN
SIGSEGV	SEGV_EXPLICITGEN
SIGILL	ILL_EXPLICITGEN

If SIGFPE is raised because of a floating-point exception, the user handler is called with one extra parameter that specifies the FPE_xxx type of the signal. If SIGSEGV, SIGILL, or the integer-related variants of SIGFPE signals (FPE_INTOVFLOW or FPE_INTDIV0) are raised as the result of a processor exception, the user handler is called with two extra parameters:

1. The SIGFPE, SIGSEGV, or SIGILL exception type (see float.h for all these types). This first parameter is the usual ANSI signal type.
2. An integer pointer into the stack of the interrupt handler that called the user-specified handler. This pointer points to a list of the processor registers saved when the exception occurred. The registers are in the same order as the parameters to an interrupt function; that is, BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS, FLAGS. To have a register value changed when the handler returns, change one of the locations in this list.

   For example, to have a new SI value on return, do something like this:

   ```
   *((int*)list_pointer + 2) = new_SI_value;
   ```

   In this way, the handler can examine and make any adjustments to the registers that you want.

The following SIGFPE-type signals can occur (or be generated). They correspond to the exceptions that the 8087 family is capable of detecting, as well as the "INTEGER DIVIDE BY ZERO" and the "INTERRUPT ON OVERFLOW" on the main CPU. (The declarations for these are in float.h.)

SIGFPE signal	Meaning
FPE_INTOVFLOW	INTO executed with OF flag set
FPE_INTDIV0	Integer divide by zero
FPE_INVALID	Invalid operation
FPE_ZERODIVIDE	Division by zero
FPE_OVERFLOW	Numeric overflow
FPE_UNDERFLOW	Numeric underflow
FPE_INEXACT	Precision
FPE_EXPLICITGEN	User program executed *raise*(SIGFPE)
FPE_STACKFAULT	Floating-point stack overflow or underflow
FPE_STACKFAULT	Stack overflow

The FPE_INTOVFLOW and FPE_INTDIV0 signals are generated by integer operations, and the others are generated by floating-point operations. Whether the floating-point exceptions are generated depends on the coprocessor control word, which can be modified with *_control87*. Denormal exceptions are handled by Borland C++ and not passed to a signal handler.

The following SIGSEGV-type signals can occur:

SEGV_BOUND	Bound constraint exception
SEGV_EXPLICITGEN	*raise*(SIGSEGV) was executed

The 8088 and 8086 processors *don't* have a bound instruction. The 186, 286, 386, and NEC V series processors *do* have this instruction. So, on the 8088 and 8086 processors, the SEGV_BOUND type of SIGSEGV signal won't occur. Borland C++ doesn't generate bound instructions, but they can be used in

inline code and separately compiled assembler routines that are linked in.

The following SIGILL-type signals can occur:

ILL_EXECUTION	Illegal operation attempted
ILL_EXPLICITGEN	*raise*(SIGILL) was executed

The 8088, 8086, NEC V20, and NEC V30 processors *do not* have an illegal operation exception. The 186, 286, 386, NEC V40, and NEC V50 processors *do* have this exception type. On 8088, 8086, NEC V20, and NEC V30 processors, the ILL_EXECUTION type of SIGILL won't occur.

When the signal type is SIGFPE, SIGSEGV, or SIGILL, a return from a signal handler is generally not advisable if the state of the 8087 is corrupt, the results of an integer division are wrong, an operation that shouldn't have overflowed did, a bound instruction failed, or an illegal operation was attempted. The only time a return is reasonable is when the handler alters the registers so that a reasonable return context exists *or* the signal type indicates that the signal was generated explicitly (for example, FPE_EXPLICITGEN, SEGV_EXPLICITGEN, or ILL_EXPLICITGEN). Generally in this case you would print an error message and terminate the program using _exit, exit, or abort. If a return is executed under any other conditions, the program's action will probably be unpredictable.

Note: Take special care when using the *signal* function in a multithread program. The SIGINT, SIGTERM, and SIGBREAK signals can be used only by the main thread (thread one) in a non-Win32 application. When one of these signals occurs, the currently executing thread is suspended, and control transfers to the signal handler (if any) set up by thread one. Other signals can be handled by any thread.

A signal handler should not use C++ run-time library functions, because a semaphore deadlock might occur. Instead, the handler should simply set a flag or post a semaphore, and return immediately.

Return Value

On success, *signal* returns a pointer to the previous handler routine for the specified signal type.

On error, *signal* returns SIG_ERR, and the external variable errno is set to EINVAL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

sin, sinl

See also Example Portability

Syntax

```
#include <math.h>
double sin(double x);
long double sinl(long double x);
```

Description

Calculates sine.

sin computes the sine of the input value. Angles are specified in radians.

sinl is the **long double** version; it takes a l**ong double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

Return Value

sin and *sinl* return the sine of the input value.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sin	+	+	+	+	+		+
sinl	+		+	+			+

sinh, sinhl

See also Example Portability

Syntax

```
#include <math.h>
double sinh(double x);
long double sinhl(long double x);
```

Description

Calculates hyperbolic sine.

sinh computes the hyperbolic sine, $(e^x - e^x)/2$.

sinl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for *sinh* and *sinhl* can be modified through the functions *_matherr* and *_matherrl*.
This function can be used with *bcd* and *complex* types.

Return Value

sinh and *sinhl* return the hyperbolic sine of *x*.

When the correct value overflows, these functions return the value HUGE_VAL (*sinh*) or _LHUGE_VAL (*sinhl*) of appropriate sign. Also, the global variable *errno* is set to ERANGE.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sinh	+	+	+	+	+		+
sinhl	+		+	+			+

sleep

Example Portability

Syntax

```
#include <dos.h>
void sleep(unsigned seconds);
```

Description

Suspends execution for an interval (seconds).

With a call to *sleep*, the current program is suspended from execution for the number of seconds specified by the argument *seconds*. The interval is accurate only to the nearest hundredth of a second or to the accuracy of the operating system clock, whichever is less accurate.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+			+

_sopen, _wsopen

See also Example Portability

Syntax

```
#include <fcntl.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>
#include <stdio.h>
int _sopen(char *path, int access, int shflag[, int mode]);
int _wsopen(wchar_t *path, int access, int shflag[, int mode]);
```

Description

Opens a shared file.

_sopen opens the file given by *path* and prepares it for shared reading or writing, as determined by *access, shflag*, and *mode*.

_wsopen is the Unicode version of *_sopen*. The Unicode version accepts a filename that is a *wchar_t* character string. Otherwise, the functions perform identically.

For *_sopen*, *access* is constructed by ORing flags bitwise from the following lists:

Read/write flags

You can use only one of the following flags:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Other access flags

You can use any logical combination of the following flags:

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer is set to the end of the file prior to each write.
O_CREA	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits as in *chmod*.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	This flag can be given to explicitly open the file in binary mode.
O_TEXT	This flag can be given to explicitly open the file in text mode.
O_NOINHERIT	The file is not passed to child programs.

Note: These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing *access*, you need to supply the *mode* argument to *_sopen* from the following symbolic constants defined in sys\stat.h.

Value of mode	Access permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read/write

shflag specifies the type of file-sharing allowed on the file *path*. Symbolic constants for *shflag* are

defined in share.h.

Value of shflag	What it does
SH_COMPAT	Sets compatibility mode.
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access
SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On success, *_sopen* returns a nonnegative integer (the file handle), and the file pointer (that marks the current position in the file) is set to the beginning of the file.

On error, it returns -1, and the global variable errno is set to

EACCES	ermission denied
EINVACC	nvalid access code
EMFILE	oo many open files
ENOENT	Path or file function not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe

See also Examples Portability

Syntax

```
#include <process.h>
#include <stdio.h>
int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnl(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);
int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char
  *envp[]);
int _wspawnle(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL,
  wchar_t *envp[]);
int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnlp(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn,
  NULL);
int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char
  *envp[]);
int _wspawnlpe(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn,
  NULL, wchar_t *envp[]);
int spawnv(int mode, char *path, char *argv[]);
int _wspawnv(int mode, wchar_t *path, wchar_t *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int _wspawnve(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
int spawnvp(int mode, char *path, char *argv[]);
int _wspawnvp(int mode, wchar_t *path, wchar_t *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
int _wspawnvpe(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
```

Note: In *spawnle, spawnlpe, spawnv, spawnve, spawnvp*, and *spawnvpe*, the last string must be NULL.

Description

The functions in the *spawn*... family create and run (execute) other files, known as child processes. There must be sufficient memory available for loading and executing a child process.

The value of *mode* determines what action the calling function (the *parent process*) takes after the *spawn*... call. The possible values of *mode* are

P_WAIT	Puts parent process on hold until child process completes execution.
P_NOWAIT	Continues to run parent process while child process runs. The child process ID is returned, so that the parent can wait for completion using *cwait* or *wait*. This mode is currently not available for 16-bit Windows or 16-bit DOS; using it generates an error value.
P_NOWAITO	Identical to P_NOWAIT except that the child process ID isn't saved by the operating system, so the parent process can't wait for it using *cwait* or *wait*.
P_DETACH	Identical to P_NOWAITO, except that the child process is executed in the background with no access to the keyboard or the display.
P_OVERLAY	Overlays child process in memory location formerly occupied by parent. Same as an *exec*... call.

path is the file name of the called child process. The *spawn...* function calls search for *path* using the standard operating system search algorithm:

- If there is no extension or no period, they search for an exact file name. If the file is not found, they search for files first with the extension EXE, then COM, and finally BAT.
- If an extension is given, they search only for the exact file name.
- If only a period is given, they search only for the file name with no extension.
- If *path* does not contain an explicit directory, *spawn...* functions that have the **p** suffix search the

current directory, then the directories set with the operating system PATH environment variable.

The suffixes *p, l*, and *v*, and *e* added to the *spawn...* "family name" specify that the named function operates with certain capabilities.

- **p** The function searches for the file in those directories specified by the PATH environment variable. Without the p suffix, the function searches only the current working directory.
- **l** The argument pointers *arg0, arg1, ..., argn* are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.
- **v** The argument pointers *argv[0], ..., arg[n]* are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.
- **e** The argument *envp* can be passed to the child process, letting you alter the environment for the child process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the *spawn...* family must have one of the two argument-specifying suffixes (either *l* or *v*). The path search and environment inheritance suffixes (*p* and *e*) are optional.

For example:

- *spawnl* takes separate arguments, searches only the current directory for the child, and passes on the parent's environment to the child.
- *spawnvpe* takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *envp* argument for altering the child's environment.

The *spawn...* functions must pass at least one argument to the child process (*arg0 or argv[0]*). This argument is, by convention, a copy of *path*. (Using a different value for this 0 argument won't produce an error.) If you want to pass an empty argument list to the child process, then *arg0* or *argv[0]* must be NULL.

Under DOS 3.x, path is available for the child process; under earlier versions, the child process cannot use the passed value of the 0 argument (*arg0* or *argv[0]*).

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1, ..., argn* point to character strings that form the new list of arguments. A mandatory null following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *envp*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *envp[]* is null. When *envp* is null, the child inherits the parents' environment settings.

The combined length of *arg0* + *arg1* + ... + *argn* (or of *argv[0]* + *argv[1]* + ... + *argv[n]*), including space characters that separate the arguments, must be less than 260 bytes for Windows (128 for DOS). Null-terminators are not counted.

When a *spawn...* function call is made, any open files remain open in the child process.

Return Value

When successful, the *spawn...* functions, where *mode* is P_WAIT, return the child process' exit status (0 for a normal termination). If the child specifically calls *exit* with a nonzero argument, its exit status can be set to a nonzero value.

If *mode* is P_NOWAIT or P_NOWAITO, the spawn functions return the process ID of the child process. The ID obtained when using P_NOWAIT can be passed to cwait.

On error, the *spawn...* functions return -1, and the global variable errno is set to one of the following values:

E2BIG	Arg list too long
EINVAL	Invalid argument
ENOENT	Path or file name not found

ENOEXEC	Exec format error
ENOMEM	Not enough memory

Examples

spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

_splitpath, _wsplitpath

See also Example Portability

Syntax

```
#include <stdlib.h>
void _splitpath(const char *path, char *drive, char *dir, char *name, char
  *ext);
void _wsplitpath(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t
  *name, wchar_t *ext);
```

Description

Splits a full path name into its components.

_splitpath takes a file's full path name (*path*) as a string in the form

X:\DIR\SUBDIR\NAME.EXT

and splits *path* into its four components. It then stores those components in the strings pointed to by *drive*, *dir*, *name*, and *ext*. (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.) The maximum sizes for these strings are given by the constants _MAX_DRIVE, _MAX_DIR, _MAX_PATH, _MAX_FNAME, and _MAX_EXT (defined in stdlib.h), and each size includes space for the null-terminator. These constants are defined in stdlib.h.

Constant	String
_MAX_PATH	*path*
_MAX_DRIVE	*drive*; includes colon (:)
_MAX_DIR	*dir*; includes leading and trailing backslashes (\)
_MAX_FNAME	name
_MAX_EXT	*ext*; includes leading dot (.)

_splitpath assumes that there is enough space to store each non-null component.

When *_splitpath* splits *path*, it treats the punctuation as follows:

- *drive* includes the colon (C:, A:, and so on).
- *dir* includes the leading and trailing backslashes (\BC\include\, \source\, and so on).
- *name* includes the file name.
- *ext* includes the dot preceding the extension (.C, .EXE, and so on).

_makepath and *_splitpath* are invertible; if you split a given *path* with *_splitpath*, then merge the resultant components with *_makepath*, you end up with *path*.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

sprintf, swprintf

See also Example Portability

Syntax

```
#include <stdio.h>
int sprintf(char *buffer, const char *format[, argument, ...]);
int swprintf(wchar_t *buffer, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a string.

Note: For details on format specifiers, see printf.

sprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a string.

sprintf applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.

Return Value

On success, *sprintf* returns the number of bytes output. The return value does not include the terminating null byte in the count.

On error, *sprintf* returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

sqrt, sqrtl

See also Example Portability

Syntax

```
#include <math.h>
double sqrt(double x);
long double sqrtl(long double x);
```

Description

Calculates the positive square root.

sqrt calculates the positive square root of the argument *x*.

sqrtl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

This function can be used with *bcd* and *complex* types.

Return Value

On success, *sqrt* and *sqrtl* return the value calculated, the square root of *x*. If *x* is real and positive, the result is positive. If *x* is real and negative, the global variable *errno* is set to

EDOM	Domain error

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sqrt	+	+	+	+	+		+
sqrtl	+		+	+			+

srand

See also Example Portability

Syntax

```
#include <stdlib.h>
void srand(unsigned seed);
```

Description

Initializes random number generator.

The random number generator is reinitialized by calling *srand* with an argument value of 1. It can be set to a new starting point by calling *srand* with a given *seed* number.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

sscanf, swscanf

See also Example Portability

Syntax

```
#include <stdio.h>
int sscanf(const char *buffer, const char *format[, address, ...]);
int swscanf(const wchar_t *buffer, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a string.

Note: For details on format specifiers, see scanf.

sscanf scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to *sscanf* in the format string pointed to by *format*. Finally, *sscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

sscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

Return Value

On success, *sscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If *sscanf* attempts to read at end-of-string, it returns EOF.

On error (If no fields were stored), it returns 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

stackavail

See also Example Portability

Syntax

```
#include <malloc.h>
size_t stackavail(void);
```

Description

Gets the amount of available stack memory.

stackavail returns the number of bytes available on the stack. This is the amount of dynamic memory that *alloca* can access.

Return Value

stackavail returns a *size_t* value indicating the number of bytes available.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_status87

Example Portability

Syntax

```
#include <float.h>
unsigned int _status87(void);
```

Description

Gets floating-point status.

_status87 gets the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

Return Value

The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by *_status87*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

stime

See also Example Portability

Syntax

```
#include <time.h>
int stime(time_t *tp);
```

Description

Sets system date and time.

stime sets the system time and date. *tp* points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970.

Return Value

stime returns a value of 0.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+				+

_stpcpy, _wcspcpy

See also Example Portability

Syntax

```
#include <string.h>
char *stpcpy(char *dest, const char *src);
wchar * _wcspcpy(wchar *dest, const wchar *src);
```

Description

Copies one string into another.

_stpcpy copies the string *src* to *dest*, stopping after the terminating null character of *src* has been reached.

Return Value

stpcpy returns a pointer to the terminating null character of *dest*.

If UNICODE is defined, *_wcspcpy* returns a pointer to the terminating null character of the **wchar_t** *dest* string.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

strcat, _fstrcat, _mbscat, wcscat

Example Portability

Syntax

```
#include <string.h>
char *strcat(char *dest, const char *src);
wchar_t *wcscat(wchar_t *dest, const wchar_t *src);
char __far * _fstrcat(char __far *dest, const char __far *src);

#include <mbstring.h>
unsigned char *_mbscat(unsigned char *dest, const unsigned char *src);
```

Description

Appends one string to another.

strcat appends a copy of *src* to the end of *dest*. The length of the resulting string is *strlen*(*dest*) + *strlen*(*src*).

Return Value

strcat returns a pointer to the concatenated strings.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strcat	+	+	+	+	+	+	+
_fstrcat	+		+				

strchr, _fstrchr, _mbschr, wcschr

See also Example Portability

Syntax

```
#include <string.h>
char *strchr(const char *s, int c);                          /* C only */
char far * far _fstrchr(const char far *s, int c)            /* C only */

const char *strchr(const char *s, int c);                    // C++ only
char *strchr( char *s, int c);                               // C++ only
wchar_t *wcschr(const wchar_t *s, int c);

const char far * far _fstrchr(const char far *s, int c);     // C++ only
char far * far _fstrchr( char far *s, int c);                // C++ only
#include <mbstring.h>
unsigned char * _mbschr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the first occurrence of a given character.

strchr scans a string in the forward direction, looking for a specific character. *strchr* finds the *first* occurrence of the character *c* in the string *s*. The null-terminator is considered to be part of the string.

For example:

```
strchr(strs,0)
```

returns a pointer to the terminating null character of the string *strs*.

Return Value

strchr returns a pointer to the first occurrence of the character *c* in *s*; if *c* does not occur in *s*, *strchr* returns null.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strchr	+	+	+	+	+	+	+
_fstrchr	+		+				

strcmp, _fstrcmp, _mbscmp, wcscmp

See also Example Portability

Syntax

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int _fstrcmp(const far char *s1, const far char *s2);
int wcscmp(const wchar_t *s1, const wchar_t *s2);

#include <mbstring.h>
int _mbscmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another.

strcmp performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return Value

If *s1* is...	return value is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strcmpi

See also Example Portability

Syntax

```
#include <string.h>
int strcmpi(const char *s1, const char *s2);
```

Description

Compares one string to another, without case sensitivity.

strcmpi performs an unsigned comparison of *s1* to *s2*, without case sensitivity (same as *stricmp*--implemented as a macro).

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routine *strcmpi* is the same as *stricmp*. *strcmpi* is implemented through a macro in string.h and translates calls from *strcmpi* to *stricmp*. Therefore, in order to use *strcmpi*, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

If s1 is...	strcmpi returns a value that is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+			+

strcoll, _stricoll, _mbscoll, _mbsicoll, wcscoll, _wcsicoll

See also Example Portability

Syntax

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);

int _stricoll(const char *s1, const char *s2);
int _wcsicoll(const wchar_t *s1, wconst_t char *s2);

#include <mbstring.h>
int _mbscoll(const unsigned char *s1, const unsigned char *s2);
int _mbsicoll(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares two strings.

strcoll compares the string pointed to by *s1* to the string pointed to by *s2*, according to the current locale's LC_COLLATE category.

_stricoll performs like *strcoll* but is not case sensitive.

Return Value

If s1 is...	functions return a value that is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

strcpy, _mbscpy, wcscpy

See also Example Portability

Syntax

```
#include <string.h>
char *strcpy(char *dest, const char *src);
wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);

#include <mbstring.h>
unsigned char *_mbscpy(unsigned char *dest, const unsigned char *src);
```

Description

Copies one string into another.

Copies string *src* to *dest*, stopping after the terminating null character has been moved.

Return Value

strcpy returns *dest*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strcspn, _fstrcspn, _mbscspn, wcscspn

See also Example Portability

Syntax

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t far *far _fstrcspn(const char far *s1, const char far *s2)

#include <mbstring.h>
size_t _mbscspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the initial segment not containing any subset of a given set of characters.

The *strcspn* functions search *s1* until any one of the characters contained in *s2* is found. The number of characters which were read in *s1* is the return value. The string termination character is not counted. Neither string is altered during the search.

Return Value

strcspn returns the length of the initial segment of string *s1* that consists entirely of characters *not* from string *s2*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strcspn	+	+	+	+	+	+	+
_fstrcspn	+		+				

_strdate, _wstrdate

See also Example Portability

Syntax

```
#include <time.h>
char *_strdate(char *buf);
wchar_t *_wstrdate(wchar_t *buf);
```

Description

Converts current date to string.

_strdate converts the current date to a string, storing the string in the buffer *buf*. The buffer must be at least 9 characters long.

The string has the form MM/DD/YY where MM, DD, and YY are all two-digit numbers representing the month, day, and year. The string is terminated by a null character.

Return Value

_strdate returns *buf*, the address of the date string.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

strdup, _fstrdup, _mbsdup, _wcsdup

See also Example Portability

Syntax

```
#include <string.h>
char *strdup(const char *s);
wchar_t *_wcsdup(const wchar_t *s);
char far *_fstrdup(const char far *s)

#include <mbstring.h>
unsigned char *_mbsdup(const wchar_t *s);
```

Description

Copies a string into a newly created location.

strdup makes a duplicate of string *s*, obtaining space with a call to *malloc*. The allocated space is (*strlen(s)* + 1) bytes long. The user is responsible for freeing the space allocated by *strdup* when it is no longer needed.

Return Value

strdup returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strdup	+	+	+	+			+
_fstrdup	+		+				

_strerror

See also Example Portability

Syntax

```
#include <string.h>
char  *_strerror(const char *s);
```

Description

Builds a customized error message.

_strerror lets you generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If *s* is null, the return value points to the most recent error message.
- If *s* is not null, the return value contains *s* (your customized error message), a colon, a space, the most-recently generated system error message, and a new line. *s* should be 94 characters or less.

Return Value

_strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to *_strerror*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

strerror

See also Example Portability

Syntax

```
#include <string.h>
char *strerror(int errnum);
```

Description

Returns a pointer to an error message string.

strerror takes an **int** parameter *errnum*, an error number, and returns a pointer to an error message string associated with *errnum*.

Return Value

strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to *strerror*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

strftime, wcsftime

See also Example Portability

Syntax

```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm
  *t);
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *fmt, const struct
  tm *t);
```

Description

Formats time for output.

strftime formats the time in the argument *t* into the array pointed to by the argument *s* according to the *fmt* specifications. All ordinary characters are copied unchanged. No more than *maxsize* characters are placed in *s*.

The time is formatted according to the current locale's LC_TIME category.

Return Value

On success, *strftime* returns the number of characters placed into *s.*

On error (if the number of characters required is greater than *maxsize*), *strftime* returns 0.

More about strftime

ANSI-defined format specifiers

POSIX-defined Format Specifiers

POSIX-defined Format Specifier Modifiers

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

strftime Format String

Consists of zero or more directives and ordinary characters. A directive consists of the % character followed by a character that determines the substitution that is to take place.

ANSI-defined Format Specifiers for strftime

See also

The following table describes the ANSI-defined specifiers for the format string used with strftime.

Format specifier	Substitutes
%%	Character %
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time
%d	Two-digit day of month (01 - 31)
%H	Two-digit hour (00 - 23)
%I	Two-digit hour (01 - 12)
%j	Three-digit day of year (001 - 366)
%m	Two-digit month as a decimal number (1 - 12)
%M	2-digit minute (00 - 59)
%p	AM or PM
%S	Two-digit second (00 - 59)
%U	Two-digit week number where Sunday is the first day of the week (00 - 53)
%w	Weekday where 0 is Sunday (0 - 6)
%W	Two-digit week number where Monday is the first day of week the week (00 - 53)
%x	Date
%X	Time
%y	Two-digit year without century (00 to 99)
%Y	Year with century
%Z	Time zone name, or no characters if no time zone

POSIX-defined Format Specifiers for strftime

See also

The following table describes the POSIX-defined specifiers for the format string used with strftime.

Note: You must define __USELOCALES__ in order to use these descriptors.

Format specifier	Substitution
%C	Century as a decimal number (00 - 99). For example, 1992 => 19
%D	Date in the format mm/dd/yy
%e	Day of the month as a decimal number in a two-digit field with leading space (1 -31)
%h	A synonym for %b
%n	Newline character
%r	12-hour time (01 - 12) format with am/pm string i.e. "%I:%M:%S %p"
%t	Tab character
%T	24-hour time (00 - 23) in the format "HH:MM:SS"
%u	Weekday as a decimal number (1 Monday - 7 Sunday)

Modifiers

strftime also supports POSIX-defined modifiers for certain specifiers. See POSIX-defined Format Specifier Modifiers.

POSIX-defined Format Specifier Modifiers for strftime

See also

The following table describes the POSIX-defined modifiers for the following format string specifiers used with strftime.

Note: You must define __USELOCALES__ in order to use these descriptors.

Descriptor modifier	Substitutes
%Od	Day of the month using alternate numeric symbols
%Oe	Day of the month using alternate numeric symbols
%OH	Hour (24 hour) using alternate numeric symbols
%OI	Hour (12 hour) using alternate numeric symbols
%Om	Month using alternate numeric symbols
%OM	Minutes using alternate numeric symbols
%OS	Seconds using alternate numeric symbols
%Ou	Weekday as a number using alternate numeric symbols
%OU	Week number of the year using alternate numeric symbols
%Ow	Weekday as number using alternate numeric symbols
%OW	Week number of the year using alternate numeric symbols
%Oy	Year (offset from %C) using alternate numeric symbols

%O modifier

When the %O modifier is used before any of the above supported numeric format descriptors (for example, `%Od`), the numeric value is converted to the corresponding ordinal string, if it exists. If an ordinal string does not exist, the basic format descriptor is used unmodified.

For example, on 4/20/94:

- %d produces 20
- %Od produces 20th

stricmp, _fstricmp, _mbsicmp, _wcsicmp

See also Example Portability

Syntax

```
#include <string.h>
int stricmp(const char *s1, const char *s2);
int _wcsicmp(const wchar_t *s1, const wchar_t *s2);
int far _fstricmp(const char far *s1, const char far *s2)

#include <mbstring.h>
int _mbsicmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another, without case sensitivity.

stricmp performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines *stricmp* and *strcmpi* are the same; *strcmpi* is implemented through a macro in string.h that translates calls from *strcmpi* to *stricmp*. Therefore, in order to use *stricmp*, you must include the header file string.h for the macro to be available.

Return Value

If s1 is...	return value is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
stricmp	+	+	+	+	+	+	+
_fstricmp	+		+				

strlen, _fstrlen, _mbslen, wcslen, _mbstrlen

Example Portability

Syntax

```
#include <string.h>
size_t strlen(const char *s);
size_t wcslen(const wchar_t *s);
size_t far _fstrlen(const char far *s)

#include <mbstring.h>
size_t _mbslen(const unsigned char *s);

#include <stdlib.h>
size_t _mbstrlen(const char *s)
```

Description

Calculates the length of a string.

strlen calculates the length of *s*.

_mbslen and *_mbstrlen* test the string argument to determine the number of multibyte characters they contain.

_mbstrlen is affected by the *LC_CTYPE* category setting as determined by the *setlocale* function. The function tests to determine whether the string argument is a valid multibyte string.

_mbslen is affected by the code page that is in use. This function doesn't test for multibyte validity.

Return Value

strlen returns the number of characters in *s*, not counting the null-terminating character.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strlen	+	+	+	+	+	+	+
_fstrlen	+		+				

strlwr, _fstrlwr, _mbslwr, _wcslwr

See also Example Portability

Syntax

```
#include <string.h>
char *strlwr(char *s);
wchar_t *_wcslwr(wchar_t *s);
char far *_fstrlwr(char far *s)

#include <mbstring.h>
unsigned char *_mbslwr(unsigned char *s);
```

Description

Converts uppercase letters in a string to lowercase.

strlwr converts uppercase letters in string *s* to lowercase according to the current locale's LC_CTYPE category. For the C locale, the conversion is from uppercase letters (*A* to *Z*) to lowercase letters (*a* to *z*). No other characters are changed.

Return Value

strlwr returns a pointer to the string *s*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strlwr	+	+	+	+	+	+	+
_fstrlwr	+		+				

strncat, _fstrncat, _mbsncat, wcsncat

Example Portability

Syntax

```
#include <string.h>
char *strncat(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t maxlen);
char far *_fstrncat(char far *dest, const char far *src, size_t maxlen);

#include <mbstring.h>
unsigned char *_mbsncat(unsigned char *dest, const unsigned char *src,
  size_t maxlen);
```

Description

Appends a portion of one string to another.

strncat copies at most *maxlen* characters of src to the end of *dest* and then appends a null character. The maximum length of the resulting string is *strlen(dest)* + *maxlen*.

These three functions behave identically and differ only with respect to the type of arguments and return types.

Return Value

strncat returns *dest*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strncat	+	+	+	+	+	+	+
_fstrncat	+		+				

strncmp, _fstrncmp, _mbsncmp, wcsncmp

See also Example Portability

Syntax

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t  maxlen);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t  maxlen);
int far _fstrncmp(const char far *s1, const char far *s2, size_t maxlen)

#include <mbstring.h>
int _mbsncmp(const unsigned char *s1, const unsigned char *s2, size_t
  maxlen);
```

Description

Compares a portion of one string to a portion of another.

strncmp makes the same unsigned comparison as *strcmp*, but looks at no more than *maxlen* characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined *maxlen* characters.

Return Value

These string comparison functions return an **int** value based on the result of comparing *s1* (or part of it) to *s2* (or part of it):

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strncmp	+	+	+	+	+	+	+
_fstrncmp	+		+				

strncmpi, wcsncmpi

Example Portability

Syntax

```
#include <string.h>
int strncmpi(const char *s1, const char *s2, size_t n);
int wcsncmpi(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strncmpi performs a signed comparison of *s1* to *s2*, for a maximum length of *n* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until *n* characters have been examined. The comparison is not case sensitive. (*strncmpi* is the same as *strnicmp*--implemented as a macro). It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines *strnicmp* and *strncmpi* are the same; *strncmpi* is implemented through a macro in string.h that translates calls from *strncmpi* to *strnicmp*. Therefore, in order to use *strncmpi*, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

If s1 is...	return value is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+				

strncpy, _fstrncpy, _mbsncpy, wcsncpy

Example Portability

Syntax

```
#include <stdio.h>
char *strncpy(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t maxlen);
char far * far _fstrncpy(char far *dest, const char far *src, size_t maxlen)

#include <mbstring.h>
unsigned char *_mbsncpy(unsigned char *dest, const unsigned char *src,
  size_t maxlen);
```

Description

Copies a given number of bytes from one string into another, truncating or padding as necessary.

strncpy copies up to *maxlen* characters from *src* into *dest*, truncating or null-padding *dest*. The target string, *dest*, might not be null-terminated if the length of *src* is *maxlen* or more.

Return Value

strncpy returns *dest*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strncpy	+	+	+	+	+	+	+
_fstrncpy	+		+				

_strnextc, _mbsnextc, _wcsnextc

Example

Syntax

```
#include <tchar.h>
unsigned int _strnextc(const char *str);

#include <mbstring.h>
unsigned int _mbsnextc (const unsigned char *str);
```

Description

These routines should be accessed by using the portable *_tcsnextc* function. The functions inspect the current character in *str*. The pointer to *str* is not advanced.

Return Value

The functions return the integer value of the character pointed to by *str*.

strnicmp, _fstrnicmp, _mbsnicmp, _wcsnicmp

Example Portability

Syntax

```
#include <string.h>
int strnicmp(const char *s1, const char *s2, size_t maxlen);
int _wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);

#include <mbstring.h>
int _mbsnicmp(const unsigned char *s1, const unsigned char *s2, size_t
  maxlen);
int far _fstrnicmp(const char far *s1, const char far *s2, size_t maxlen)
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strnicmp performs a signed comparison of *s1* to *s2*, for a maximum length of *maxlen* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

Return Value

If s1 is...	return value is...
less than *s2*	< 0
the same as *s2*	== 0
greater than *s2*	> 0

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strnicmp	+		+	+			+
_fstrnicmp	+		+				

strnset, _fstrnset, _mbsnset, _wcsnset

Example Portability

Syntax

```
#include <string.h>
char *strnset(char *s, int ch, size_t n);
wchar_t *_wcsnset(wchar_t *s, wchar_t ch, size_t n);
char far * far _fstrnset(char far *s, int ch, size_t n)

#include <mbstring.h>
unsigned char *_mbsnset(unsigned char *s, unsigned int ch, size_t n);
```

Description

Sets a specified number of characters in a string to a given character.

strnset copies the character *ch* into the first *n* bytes of the string *s*. If *n* > *strlen(s)*, then *strlen(s)* replaces *n*. It stops when *n* characters have been set, or when a null character is found.

Return Value

Each of these functions return *s*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strnset	+		+	+			+
_fstrnset	+		+				

strpbrk, _fstrpbrk, _mbspbrk, wcspbrk

Example Portability

Syntax

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);          /* C only */
char far *far _fstrpbrk(const char far *s1,
  const char far*s2)                                    /* C only */
const char *strpbrk(const char *s1, const char *s2);    // C++ only
char *strpbrk(char *s1, const char *s2);                // C++ only
const char far *far _fstrpbrk(const char far *s1,
  const char far *s2);                                  // C++ only
char far * far _fstrpbrk(char far *s1,
                         const char far *s2);           // C++ only
wchar_t * wcspbrk(const wchar_t *s1, const wchar_t *s2);

#include <mbstring.h>
unsigned char *_mbspbrk(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first occurrence of any character from a given set.

strpbrk scans a string, *s1*, for the first occurrence of any character appearing in *s2*.

Return Value

strpbrk returns a pointer to the first occurrence of any of the characters in *s2*. If none of the *s2* characters occur in *s1*, *strpbrk* returns null.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strpbrk	+	+	+	+	+	+	+
_fstrpbrk	+		+				

strrchr, _fstrrchr, _mbsrchr, wcsrchr

See also Example Portability

Syntax

```
char *strrchr(const char *s, int c);                    /* C only */
char far * far _fstrrchr(const char far *s, int c)      /* C only */

const char *strrchr(const char *s, int c);              // C++ only
char *strrchr(char *s, int c);                          // C++ only
const char *_fstrrchr(const char far *s, int c);        // C++ only
char *_fstrrchr(char far *s, int c);                    // C++ only
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);

#include <mbstring.h>
unsigned char * _mbsrchr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the last occurrence of a given character.

strrchr scans a string in the reverse direction, looking for a specific character. *strrchr* finds the *last* occurrence of the character *c* in the string *s*. The null-terminator is considered to be part of the string.

Return Value

Each of the functions return a pointer to the last occurrence of the character *c*. If *c* does not occur in *s*, the functions return null.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strrchr	+	+	+	+	+	+	+
_fstrrchr	+		+				

strrev, _fstrrev, _mbsrev, _wcsrev

Example Portability

Syntax

```
#include <string.h>
char *strrev(char *s);
wchar_t *_wcsrev(wchar_t *s);
char far * far _fstrrev(char far *s)

#include <mbstring.h>
unsigned char *_mbsrev(unsigned char *s);
```

Description

Reverses a string.

strrev changes all characters in a string to reverse order, except the terminating null character. (For example, it would change *string\0* to *gnirts\0*.)

Return Value

These functions return a pointer to the reversed string.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strrev	+		+	+			+
_fstrrev	+		+				

strset, _fstrset, _mbsset, _wcsset

See also Example Portability

Syntax

```
#include <string.h>
char *strset(char *s, int ch);
wchar_t *_wcsset(wchar_t *s, wchar_t ch);
char far * far _fstrset(char far *s, int ch)

#include <mbstring.h>
unsigned char *_mbsset(unsigned char *s, unsigned int ch);
```

Description

Sets all characters in a string to a given character.

strset sets all characters in the string *s* to the character *ch*. It quits when the terminating null character is found.

Return Value

Each of these functions return *s*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strset	+		+	+			+
_fstrset	+		+				

strspn, _fstrspn, _mbsspn, wcsspn

See also Example Portability

Syntax

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
size_t far _fstrspn(const char far *s1, const char far *s2)

#include <mbstring.h>
size_t _mbsspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first segment that is a subset of a given set of characters.

strspn finds the initial segment of string *s1* that consists entirely of characters from string *s2*.

Return Value

Each of these functions return the length of the initial segment of *s1* that consists entirely of characters from *s2*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strspn	+	+	+	+	+	+	+
_fstrspn	+		+				

strstr, _fstrstr, _mbsstr, wcsstr

Example Portability

Syntax

```
#include <string.h>
char *strstr(const char *s1, const char *s2);                /* C only */
char far * far _fstrstr(const char far *s1,
  const char far*s2);                                        /* C only */
const char *strstr(const char *s1, const char *s2);          // C++ only
char *strstr(char *s1, const char *s2);                      // C++ only
wchar_t * wcsstr(const wchar_t *s1, const wchar_t *s2);
const char far *far _fstrstr(const char far *s1,const char far *s2);
  // C++ only
char far * far _fstrstr(char far *s1, const char far *s2);   // C++ only
#include <mbstring.h>
unsigned char * _mbsstr(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the occurrence of a given substring.

strstr scans *s1* for the first occurrence of the substring *s2*.

Return Value

strstr returns a pointer to the element in *s1*, where *s2* begins (points to *s2* in *s1*). If *s2* does not occur in *s1*, *strstr* returns null.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strstr	+	+	+	+	+	+	+
_fstrstr	+		+				

_strtime, _wstrtime

See also Example Portability

Syntax

```
#include <time.h>
char *_strtime(char *buf);
wchar_t *_wstrtime(wchar_t *buf);
```

Description

Converts current time to string.

_strtime converts the current time to a string, storing the string in the buffer *buf*. The buffer must be at least 9 characters long.

The string has the following form:

```
HH:MM:SS
```

where HH, MM, and SS are all two-digit numbers representing the hour, minute, and second, respectively. The string is terminated by a null character.

Return Value

_strtime returns *buf*, the address of the time string.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

strtod, _strtold, wcstod, _wcstold

See also Example Portability

Syntax

```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
double wcstod(const wchar_t *s, wchar_t **endptr);
long double _strtold(const char *s, char **endptr);
long double _wcstold(const wchar_t *s, wchar_t **endptr);
```

Description

Convert a string to a **double** or **long double** value.

strtod converts a character string, *s*, to a **double** value. *s* is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

where:

[ws]	=	optional whitespace
[sn]	=	optional sign (+ or -)
[ddd]	=	optional digits
[fmt]	=	optional *e* or *E*
[.]	=	optional decimal point

strtod also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for Not-a-Number.

For example, here are some character strings that *strtod* can convert to **double**:

```
+ 1231.1981 e-1
502.85E2
+ 2010.952
```

strtod stops reading the string at the first character that cannot be interpreted as an appropriate part of a **double** value.

If *endptr* is not null, *strtod* sets **endptr* to point to the character that stopped the scan (**endptr* = &*stopper*). *endptr* is useful for error detection.

_strtold is the l**ong double** version; it converts a string to a **long double** value.

Return Value

These functions return the value of *s* as a **double** (*strtod*) or a **long double** (*_strtold*). In case of overflow, they return plus or minus HUGE_VAL (*strtod*) or _LHUGE_VAL (*_strtold*).

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strtod	+	+	+	+	+	+	+
_strtold	+		+	+			+

strtok, _fstrtok, _mbstok, wcstok

Example Portability

Syntax

```
#include <string.h>
char *strtok(char *s1, const char *s2);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2);
char far * far _fstrtok(char far *s1, const char far *s2)

#include <mbstring.h>
unsigned char *_mbstok(unsigned char *s1, const unsigned char *s2);
```

Description

Searches one string for tokens, which are separated by delimiters defined in a second string.

strtok considers the string *s1* to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string *s2*.

The first call to *strtok* returns a pointer to the first character of the first token in *s1* and writes a null character into *s1* immediately following the returned token. Subsequent calls with null for the first argument will work through the string *s1* in this way, until no tokens remain.

The separator string, *s2*, can be different from call to call.

Note: Calls to *strtok* cannot be nested with a function call that also uses *strtok*. Doing so will causes an endless loop.

Return Value

strtok returns a pointer to the token found in *s1*. A NULL pointer is returned when there are no more tokens.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strtok	+	+	+	+	+	+	+
_fstrtok	+		+				

strtol, wcstol

See also Example Portability

Syntax

```
#include <stdlib.h>
long strtol(const char *s, char **endptr, int radix);
long wcstol(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to a **long** value.

strtol converts a character string, *s*, to a **long** integer value. *s* is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:

```
[ws] [sn] [0] [x] [ddd]
```

where:

[ws] = optional whitespace
[sn] = optional sign (+ or -)
[0] = optional zero (0)
[x] = optional x or X
[ddd] = optional digits

strtol stops reading the string at the first character it doesn't recognize.

If *radix* is between 2 and 36, the long integer is expressed in base *radix*. If *radix* is 0, the first few characters of *s* determine the base of the value being converted.

First character	Second character	String interpreted as...
0	1 - 7	Octal
0	x or X	Hexadecimal
1 - 9		Decimal

If *radix* is 1, it is considered to be an invalid value. If *radix* is less than 0 or greater than 36, it is considered to be an invalid value.

Any invalid value for *radix* causes the result to be 0 and sets the next character pointer **endptr* to the starting string pointer.

If the value in *s* is meant to be interpreted as octal, any character other than 0 to 7 will be unrecognized.

If the value in *s* is meant to be interpreted as decimal, any character other than 0 to 9 will be unrecognized.

If the value in *s* is meant to be interpreted as a number in any other base, then only the numerals and letters used to represent numbers in that base will be recognized. (For example, if *radix* equals 5, only 0 to 4 will be recognized; if *radix* equals 20, only 0 to 9 and *A* to *J* will be recognized.)

If *endptr* is not null, *strtol* sets **endptr* to point to the character that stopped the scan (**endptr* = &*stopper*).

Return Value

strtol returns the value of the converted string, or 0 on error.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

strtoul, wcstoul

See also Example Portability

Syntax

```
#include <stdlib.h>
unsigned long strtoul(const char *s, char **endptr, int radix);
unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to an **unsigned long** in the given radix.

strtoul operates the same as *strtol*, except that it converts a string *str* to an **unsigned long** value (where *strtol* converts to a **long**). Refer to the entry for *strtol* for more information.

Return Value

strtoul returns the converted value, an **unsigned long**, or 0 on error.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

strupr, _fstrupr, _mbsupr, _wcsupr

See also Example Portability

Syntax

```
#include <string.h>
char *strupr(char *s);
wchar_t *_wcsupr(wchar_t *s);
char far * far _fstrupr(char far *s)

#include <mbstring.h>
unsigned char *_mbsupr(unsigned char *s);
```

Description

Converts lowercase letters in a string to uppercase.

strupr converts lowercase letters in string *s* to uppercase according to the current locale's LC_CTYPE category. For the default C locale, the conversion is from lowercase letters *(a* to *z*) to uppercase letters (*A* to *Z*). No other characters are changed.

Return Value

strupr returns *s.*

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strupr	+		+	+			+
_fstrupr	+		+				

strxfrm, wcsxfrm

See also Example Portability

Syntax

```
#include<string.h>
size_t strxfrm(char *target, const char *source, size_t n);
size_t wcsxfrm(wchar_t *target, const wchar_t *source, size_t n);
```

Description

Transforms a portion of a string to a specified collation.

strxfrm transforms the string pointed to by *source* into the string *target* for no more than *n* characters. The transformation is such that if the *strcmp* function is applied to the resulting strings, its return corresponds with the return values of the *strcoll* function.

No more than *n* characters, including the terminating null character, are copied to *target*.

strxfrm transforms a character string into a special string according to the current locale's LC_COLLATE category. The special string that is built can be compared with another of the same type, byte for byte, to achieve a locale-correct collation result. These special strings, which can be thought of as keys or tokenized strings, are not compatible across the different locales.

The tokens in the tokenized strings are built from the collation weights used by *strcoll* from the active locale's collation tables.

Processing stops only after all levels have been processed for the character string or the length of the tokenized string is equal to the maxlen parameter.

All redundant tokens are removed from each level's set of tokens.

The tokenized string buffer must be large enough to contain the resulting tokenized string. The length of this buffer depends on the size of the character string, the number of collation levels, the rules for each level and whether there are any special characters in the character string. Certain special characters can cause extra character processing of the string resulting in more space requirements. For example, the French character "oe" will take double the space for itself because in some locales, it expands to collation weights for each level. Substrings that have substitutions will also cause extra space requirements.

There is no safe formula to determine the required string buffer size, but at least (levels * string length) are required.

Return Value

Number of characters copied not including the terminating null character. If the value returned is greater than or equal to *n*, the content of *target* is indeterminate.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

swab

Example Portability

Syntax

```
#include <stdlib.h>
void swab(char *from, char *to, int nbytes);
```

Description

Swaps bytes.

swab copies *nbytes* bytes from the *from* string to the *to* string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. *nbytes* should be even.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

system, _wsystem

See also Example Portability

Syntax

```
#include <stdlib.h>
int system(const char *command);
int _wsystem(const wchar_t *command);
```

Description

Issues an operating system command.

system invokes the operating system command processor to execute an operating system command, batch file, or other program named by the string *command*, from inside an executing C program.

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

The COMSPEC environment variable is used to find the command processor program file, so that file need not be in the current directory.

Return Value

If *command* is a NULL pointer, *system* returns nonzero if a command processor is available.

If *command* is not a NULL pointer, *system* returns 0 if the command processor was successfully started.

If an error occurred, a -1 is returned and errno is set to one of the following:

ENOENT	Path or file function not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+			+

tan, tanl

See also Example Portability

Syntax

```
#include <math.h>
double tan(double x);
long double tanl(long double x);
```

Description

Calculates the tangent.

tan calculates the tangent. Angles are specified in radians.

tanl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these routines can be modified through the functions _matherr and _matherrl.

This function can be used with bcd and complex types.

Return Value

tan and *tanl* return the tangent of *x*, *sin(x)/cos(x)*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
tan	+	+	+	+	+	+	+
tanl	+		+	+			+

tanh, tanhl

See also Example Portability

Syntax

```
#include <math.h>
double tanh(double x);
long double tanhl(long double x);
```

Description

Calculates the hyperbolic tangent.

tanh computes the hyperbolic tangent, *sinh(x)/cosh(x)*.

tanhl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

Return Value

tanh and *tanhl* return the hyperbolic tangent of *x*.

Portability

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
tanh	+	+	+	+	+	+	+
tanhl	+		+	+			+

tell

See also Example Portability

Syntax

```
#include <io.h>
long tell(int handle);
```

Description

Gets the current position of a file pointer.

tell gets the current position of the file pointer associated with *handle* and expresses it as the number of bytes from the beginning of the file.

Return Value

tell returns the current file pointer position. A return of -1 (**long**) indicates an error, and the global variable errno is set to

EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_tempnam, _wtempnam

See also Example Portability

Syntax

```
#include <stdio.h>
char *_tempnam(char *dir, char *prefix)
wchar_t *_wtempnam(wchar_t *dir, wchar_t *prefix)
```

Description

Creates a unique file name in specified directory.

The *_tempnam* function accepts single-byte or multibyte string arguments.

The *_tempnam* function creates a unique file name in arbitrary directories. The unique file is not actually created; *_tempnam* only verifies that it does not currently exist. It attempts to use the following directories, in the order shown, when creating the file name:

- The directory specified by the TMP environment variable.
- The *dir* argument to *_tempnam*.
- The *P_tmpdir* definition in stdio.h. If you edit stdio.h and change this definition, *_tempnam* will *not* use the new definition.
- The current working directory.

If any of these directories is NULL, or undefined, or does not exist, it is skipped.

The *prefix* argument specifies the first part of the file name; it cannot be longer than 5 characters, and cannot contain a period (.). A unique file name is created by concatenating the directory name, the *prefix*, and 6 unique characters. Space for the resulting file name is allocated with *malloc*; when this file name is no longer needed, the caller should call *free* to free it.

If you do create a temporary file using the name constructed by *_tempnam*, it is your responsibility to delete the file name (for example, with a call to *remove*). It is not deleted automatically. (*tmpfile* does delete the file name.)

Return Value

If *_tempnam* is successful, it returns a pointer to the unique temporary file name, which the caller can pass to *free* when it is no longer needed. Otherwise, if *_tempnam* cannot create a unique file name, it returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

textattr

See also Example Portability

Syntax

```
#include <conio.h>
void textattr(int newattr);
```

Description

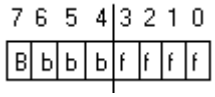
Sets text attributes.

Note: Do not use this function for Win32s or Win32 GUI applications.

textattr lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with *textcolor* and *textbackground*.)

This function does not affect any characters currently onscreen; it affects only those characters displayed by functions (such as *cprintf*) performing text mode, direct video output *after* this function is called.

The color information is encoded in the *newattr* parameter as follows:



In this 8-bit *newattr* parameter:

- *ffff* is the 4-bit foreground color (0 to 15).
- *bbb* is the 3-bit background color (0 to 7).
- *B* is the blink-enable bit.

If the blink-enable bit is on, the character blinks. This can be accomplished by adding the constant BLINK to the attribute.

If you use the symbolic color constants defined in conio.h for creating text attributes with *textattr*, note the following limitations on the color you select for the background:

- You can select only one of the first eight colors for the background.
- You must shift the selected background color left by 4 bits to move it into the correct bit positions.

These symbolic constants are listed in the following table:

Symbolic constant	Numeric value	Foreground or background
BLACK	0	Both
BLUE	1	Both
GREEN	2	Both
CYAN	3	Both
RED	4	Both
MAGENTA	5	Both
BROWN	6	Both
LIGHTGRAY	7	Both
DARKGRAY	8	Foreground only
LIGHTBLUE	9	Foreground only
LIGHTGREEN	10	Foreground only
LIGHTCYAN	11	Foreground only
LIGHTRED	12	Foreground only
LIGHTMAGENTA	13	Foreground only
YELLOW	14	Foreground only

WHITE	15	Foreground only
BLINK	128	Foreground only

Return Value
None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

textbackground

See also Example Portability

Syntax

```
#include <conio.h>
void textbackground(int newcolor);
```

Description

Selects new text background color.

Note: Do not use this function for Win32s or Win32 GUI applications.

textbackground selects the background color. This function works for functions that produce output in text mode directly to the screen. *newcolor* selects the new background color. You can set *newcolor* to an integer from 0 to 7, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called *textbackground*, all subsequent functions using direct video output (such as *cprintf*) will use *newcolor*. *textbackground* does not affect any characters currently onscreen.

The following table lists the symbolic constants and the numeric values of the allowable colors:

Symbolic constant	Numeric value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

textcolor

See also Example Portability

Syntax

```
#include <conio.h>
void textcolor(int newcolor);
```

Description

Selects new character color in text mode.

Note: Do not use this function for Win32s or Win32 GUI applications.

textcolor selects the foreground character color. This function works for the console output functions. *newcolor* selects the new foreground color. You can set *newcolor* to an integer as given in the table below, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called *textcolor*, all subsequent functions using direct video output (such as *cprintf*) will use *newcolor*. *textcolor* does not affect any characters currently onscreen.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

Symbolic constant	Numeric value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

You can make the characters blink by adding 128 to the foreground color. The predefined constant BLINK exists for this purpose.

For example:

```
textcolor(CYAN + BLINK);
```

Note: Some monitors do not recognize the intensity signal used to create the eight "light" colors (8-15). On such monitors, the light colors are displayed as their "dark" equivalents (0-7). Also, systems that do not display in color can treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, and so on). Exactly what you'll see on such systems depends on your hardware.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

textmode

See also Example Portability

Syntax

```
#include <conio.h>
void textmode(int newmode);
```

Description

Puts screen in text mode.

Note: Do not use this function for Win32s or Win32 GUI applications.

textmode selects a specific text mode.

You can give the text mode (the argument *newmode*) by using a symbolic constant from the enumeration type *text_modes* (defined in conio.h).

The most commonly used *text_modes* type constants and the modes they specify are given in the following table. Some additional values are defined in conio.h.

Symbolic Constant	Text Mode
LASTMODE	Previous text mode
BW40	Black and white, 40 columns
C40	Color, 40 columns
BW80	Black and white, 80 columns
C80	Color, 80 columns
MONO	Monochrome, 80 columns
C4350	EGA 43-line and VGA 50-line modes

When *textmode* is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to *normvideo*.

Specifying LASTMODE to *textmode* causes the most recently selected text mode to be reselected.

textmode should be used only when the screen or window is in text mode (presumably to change to a different text mode). This is the only context in which *textmode* should be used. When the screen is in graphics mode, use *restorecrtmode* instead to escape temporarily to text mode.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

time

See also Example Portability

Syntax

```
#include <time.h>
time_t time(time_t *timer);
```

Description

Gets time of day.

time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by *timer*, provided that *timer* is not a NULL pointer.

Return Value

time returns the elapsed time in seconds.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

tmpfile

See also Example Portability

Syntax

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

Opens a "scratch" file in binary mode.

tmpfile creates a temporary binary file and opens it for update (*w* + *b*). If you do not change the directory after creating the temporary file, the file is automatically removed when it's closed or when your program terminates.

Return Value

tmpfile returns a pointer to the stream of the temporary file created. If the file can't be created, *tmpfile* returns NULL.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

tmpnam, _wtmpnam

See also Example Portability

Syntax

```
#include <stdio.h>
char *tmpnam(char *s);
wchar_t *_wtmpnam(wchar_t *s);
```

Description

Creates a unique file name.

tmpnam creates a unique file name, which can safely be used as the name of a temporary file. *tmpnam* generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65,535.

The parameter to *tmpnam*, *s*, is either null or a pointer to an array of at least *L_tmpnam* characters. *L_tmpnam* is defined in stdio.h. If *s* is NULL, *tmpnam* leaves the generated temporary file name in an internal static object and returns a pointer to that object. If *s* is not NULL, *tmpnam* overwrites the internal static object and places its result in the pointed-to array, which must be at least *L_tmpnam* characters long, and returns *s*.

If you do create such a temporary file with *tmpnam*, it is your responsibility to delete the file name (for example, with a call to *remove*). It is not deleted automatically. (*tmpfile* does delete the file name.)

Return Value

If *s* is null, *tmpnam* returns a pointer to an internal static object. Otherwise, *tmpnam* returns *s*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

toascii

Example Portability

Syntax

```
#include <ctype.h>
int toascii(int c);
```

Description

Translates characters to ASCII format.

toascii is a macro that converts the integer *c* to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127.

Return Value

toascii returns the converted value of *c*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

_tolower

Example Portability

Syntax

```
#include <ctype.h>
int _tolower(int ch);
```

Description

_tolower is a macro that does the same conversion as *tolower*, except that it should be used only when *ch* is known to be uppercase (*AZ*).

To use ***_tolower***, you must include ctype.h.

Return Value

_tolower returns the converted value of *ch* if it is uppercase; otherwise, the result is undefined.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

tolower, _mbctolower, towlower

Example Portability

Syntax

```
#include <ctype.h>
int tolower(int ch);
int towlower(wint_t ch);  // Unicode version

#include <mbstring.h>
unsigned int _mbctolower(unsigned int c);
```

Description

Translates characters to lowercase.

tolower is a function that converts an integer *ch* (in the range EOF to 255) to its lowercase value (*a* to *z*; if it was uppercase, *A* to *Z*). All others are left unchanged.

Return Value

tolower returns the converted value of *ch* if it is uppercase; it returns all others unchanged.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_toupper

Example Portability

Syntax

```
#include <ctype.h>
int _toupper(int ch);
```

Description

Translates characters to uppercase.

_toupper is a macro that does the same conversion as toupper, except that it should be used only when *ch* is known to be lowercase (*a* to *z*).

To use *_toupper*, you must include ctype.h.

Return Value

_toupper returns the converted value of *ch* if it is lowercase; otherwise, the result is undefined.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

toupper, _mbctoupper, towupper

Example Portability

Syntax

```
#include <ctype.h>
int toupper(int ch);
int towupper(wint_t ch);  // Unicode version

#include <mbstring.h>
unsigned int _mbctoupper(unsigned int c);
```

Description

Translates characters to uppercase.

toupper is a function that converts an integer *ch* (in the range EOF to 255) to its uppercase value (*A* to *Z*; if it was lowercase, *a* to *z*). All others are left unchanged.

towupper is the Unicode version of *toupper*. It is available when Unicode is defined.

Return Value

toupper returns the converted value of *ch* if it is lowercase; it returns all others unchanged.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_tzset, _wtzset

See also Example Portability

Syntax

```
#include <time.h>
void _tzset(void)
void _wtzset(void)
```

Description

Sets value of global variables _daylight, _timezone, and _tzname.

_tzset is available on XENIX systems.

_tzset sets the *_daylight, _timezone*, and *_tzname* global variables based on the environment variable *TZ*. *_wtzset* sets the *_daylight, _timezone*, and *_wtzname* global variables. The library functions *ftime* and *localtime* use these global variables to adjust Greenwich Mean Time (GMT) to the local time zone. The format of the *TZ* environment string is:

```
TZ = zzz[+/-]d[d][lll]
```

where *zzz* is a three-character string representing the name of the current time zone. All three characters are required. For example, the string "PST" could be used to represent Pacific standard time.

[+/-]*d*[*d*] is a required field containing an optionally signed number with 1 or more digits. This number is the local time zone's difference from GMT in hours. Positive numbers adjust westward from GMT. Negative numbers adjust eastward from GMT. For example, the number 5 = EST, +8 = PST, and -1 = continental Europe. This number is used in the calculation of the global variable *_timezone*. *_timezone* is the difference in seconds between GMT and the local time zone.

lll is an optional three-character field that represents the local time zone, daylight saving time. For example, the string "PDT" could be used to represent pacific daylight saving time. If this field is present, it causes the global variable *_daylight* to be set nonzero. If this field is absent, *_daylight* is set to zero.

If the *TZ* environment string isn't present or isn't in the preceding form, a default *TZ* = "EST5EDT" is presumed for the purposes of assigning values to the global variables *_daylight, _timezone*, and *_tzname*. On a Win32 system, none of these global variables are set if *TZ* is null.

The global variables *_tzname*[0] and *_wtzname*[1] point to a three-character string with the value of the time-zone name from the *TZ* environment string. *_tzname*[1] and *_wtzname*[1] point to a three-character string with the value of the daylight saving time-zone name from the *TZ* environment string. If no daylight saving name is present, *_tzname*[1] and *_wtzname*[1] point to a null string.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

ultoa, _ultow

See also Example Portability

Syntax

```
#include <stdlib.h>
char *ultoa(unsigned long value, char *string, int radix);
wchar_t *_ultow(unsigned long value, wchar_t *string, int radix);
```

Description

Converts an **unsigned long** to a string.

ultoa converts *value* to a null-terminated string and stores the result in *string. value* is an **unsigned long**.

radix specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. *ultoa* performs no overflow checking, and if *value* is negative and *radix* equals 10, it does not set the minus sign.

Note: The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). *ultoa* can return up to 33 bytes.

Return Value

ultoa returns *string*.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

umask

See also Example Portability

Syntax

```
#include <io.h>
unsigned umask(unsigned mode);
```

Description

Sets file read/write permission mask.

The *umask* function sets the access permission mask used by *open* and *creat*. Bits that are set in *mode* will be cleared in the access permission of files subsequently created by *open* and *creat*.

The *mode* can have one of the following values, defined in sys\stat.h:

Value of mode	Access permission
S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

The previous value of the mask. There is no error return.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

ungetc, ungetwc

See also Example Portability

Syntax

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

Pushes a character back into input stream.

Note: Do not use this function for Win32s or Win32 GUI applications.

ungetc pushes the character *c* back onto the named input *stream*, which must be open for reading. This character will be returned on the next call to *getc* or *fread* for that *stream*. One character can be pushed back in all situations. A second call to *ungetc* without a call to *getc* will force the previous character to be forgotten. A call to *fflush, fseek, fsetpos,* or *rewind* erases all memory of any pushed-back characters.

Return Value

On success, *ungetc* returns the character pushed back.

On eror, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ungetch

See also Example Portability

Syntax

```
#include <conio.h>
int ungetch(int ch);
```

Description

Pushes a character back to the keyboard buffer.

Note: Do not use this function for Win32s or Win32 GUI applications.

ungetch pushes the character *ch* back to the console, causing *ch* to be the next character read. The *ungetch* function fails if it is called more than once before the next read.

Return Value

On success, *ungetch* returns the character *ch*.

On error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+			+

unixtodos

See also Example Portability

Syntax

```
#include <dos.h>
void unixtodos(long time, struct date *d, struct time *t);
```

Description

Converts date and time from UNIX to DOS format.

unixtodos converts the UNIX-format time given in *time* to DOS format and fills in the *date* and *time* structures pointed to by *d* and *t*.

time must not represent a calendar time earlier than Jan. 1, 1980 00:00:00.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_unlink, _wunlink

See also Example Portability

Syntax

```
#include <io.h>
int _unlink(const char *filename);
int _wunlink(const wchar_t *filename);
```

Description

Deletes a file.

_unlink deletes a file specified by *filename*. Any drive, path, and file name can be used as a *filename*. Wildcards are not allowed.

Read-only files cannot be deleted by this call. To remove read-only files, first use chmod or _rtl_chmod to change the read-only attribute.

Note: If the file is open, it must be closed before unlinking it.

_wunlink is the Unicode version of *_wunlink*. The Unicode version accepts a filename that is a *wchar_t* character string. Otherwise, the functions perform identically.

Return Value

On success, *_unlink* returns 0.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

unlock

See also Example Portability

Syntax

```
#include <io.h>
int unlock(int handle, long offset, long length);
```

Description

Releases file-sharing locks.

unlock provides an interface to the operating system file-sharing mechanism. *unlock* removes a lock previously placed with a call to *lock*. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing.

Return Value

On success, *unlock* returns 0

O error, it returns -1.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

_utime, _wutime

See also Example Portability

Syntax

```
#include <utime.h>
int _utime(char *path, struct utimbuf *times);
int _wutime(wchar_t *path, struct _utimbuf *times);
```

Description

Sets file time and date.

_utime sets the modification time for the file *path*. The modification time is contained in the *utimbuf* structure pointed to by *times*. This structure is defined in utime.h, and has the following format:

```
struct utimbuf {
   time_t   actime;    /* access time */
   time_t   modtime;   /* modification time */
   };
```

The FAT (file allocation table) file system supports only a modification time; therefore, on FAT file systems *_utime* ignores *actime* and uses only *modtime* to set the file's modification time.

If *times* is NULL, the file's modification time is set to the current time.

_wutime is the Unicode version of *_utime*. The Unicode version accepts a filename that is a *wchar_t* character string. Otherwise, the functions perform identically.

Return Value

On sucess, *_utime* returns 0.

On error, it returns -1, and sets the global variable errno to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

va_arg, va_end, va_start

See also Example Portability

Syntax

```
#include <stdarg.h>
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Description

Implement a variable argument list.

Some C functions, such as *vfprintf* and *vprintf*, take variable argument lists in addition to taking a number of fixed (known) parameters. The *va_arg, va_end*, and *va_start* macros provide a portable way to access these argument lists. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed.

The header file stdarg.h declares one type (**va_list**) and three macros (*va_start, va_arg*, and *va_end*).

- **va_list**: This array holds information needed by *va_arg* and *va_end*. When a called function takes a variable argument list, it declares a variable *ap* of type **va_list**.
- *va_start*: This routine (implemented as a macro) sets *ap* to point to the first of the variable arguments being passed to the function. *va_start* must be used before the first call to *va_arg* or *va_end*.
- *va_start* takes two parameters: *ap* and *lastfix*. (*ap* is explained under *va_list* in the preceding paragraph; *lastfix* is the name of the last fixed parameter being passed to the called function.)
- *va_arg*: This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable *ap* to *va_arg* should be the same *ap* that *va_start* initialized.

  Note: Because of default promotions, you cannot use **char, unsigned char**, or **float** types with *va_arg*.

  The first time *va_arg* is used, it returns the first argument in the list. Each successive time *va_arg* is used, it returns the next argument in the list. It does this by first dereferencing *ap*, and then incrementing *ap* to point to the following item. *va_arg* uses the *type* to both perform the dereference and to locate the following item. Each successive time *va_arg* is invoked, it modifies *ap* to point to the next argument in the list.
- *va_end*: This macro helps the called function perform a normal return. *va_end* might modify *ap* in such a way that it cannot be used unless *va_start* is recalled. *va_end* should be called after *va_arg* has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return Value

va_start and *va_end* return no values; *va_arg* returns the current argument in the list (the one that *ap* is pointing to).

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

vfprintf, vfwprintf

See also Example Portability

Syntax

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arglist);
int vfwprintf(FILE *stream, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a stream.

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Printf Format Specifiers.

vfprintf accepts a pointer to a series of arguments, applies to each argument a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Return Value

On success, *vfprintf* returns the number of bytes output.

On error, it returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

vfscanf

See also Example Portability

Syntax

```
#include <stdio.h>
int vfscanf(FILE *stream
const char *format
va_list arglist);
```

Description

Scans and formats input from a stream.

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Scanf Format Specifiers.

vfscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to *vfscanf* in the format string pointed to by *format*. Finally *vfscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields. *vfscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character or it might terminate entirely for a number of reasons. See *scanf* for a discussion of possible causes.

Return Value

vfscanf returns the number of input fields successfully scanned converted and stored; the return value does not include scanned fields that were not stored. If no fields were stored the return value is 0.

If *vfscanf* attempts to read at end-of-file the return value is EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

vprintf, vfwprintf

See also Example Portability

Syntax

```
#include <stdarg.h>
int vprintf(const char *format, va_list arglist);
int vwprintf(const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to stdout.

Note: Do not use this function for Win32s or Win32 GUI applications.

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Printf Format Specifiers.

vprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to stdout. There must be the same number of format specifiers as arguments.

Note: When you use the SS!=DS flag in 16-bit applications, *vprintf* assumes that the address being passed is in the SS segment.

Return Value

vprint returns the number of bytes output. In the event of error, *vprint* returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+		+	+		+

vscanf

See also Example Portability

Syntax

```
#include <stdarg.h>
int vscanf(const char *format, va_list arglist);
```

Description

Scans and formats input from stdin.

Note: Do not use this function for Win32s or Win32 GUI applications.

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Scanf Format Specifiers.

vscanf scans a series of input fields, one character at a time, reading from stdin. Then each field is formatted according to a format specifier passed to *vscanf* in the format string pointed to by *format*. Finally, *vscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

vscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

Return Value

vscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *vscanf* attempts to read at end-of-file, the return value is EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

vsprintf, vswprintf

See also Example Portability

Syntax

```
#include <stdarg.h>
int vsprintf(char *buffer, const char *format, va_list arglist);
int vswprintf(wchar_t *buffer, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a string.

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Printf Format Specifiers.

vsprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a string. There must be the same number of format specifiers as arguments.

Return Value

vsprintf returns the number of bytes output. In the event of error, *vsprintf* returns EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

vsscanf

See also Example Portability

Syntax

```
#include <stdarg.h>
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

Description

Scans and formats input from a stream.

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Scanf Format Specifiers.

vsscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to *vsscanf* in the format string pointed to by *format*. Finally, *vsscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

vsscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vsscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *vsscanf* attempts to read at end-of-string, the return value is EOF.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

wait

See also Portability

Syntax

```
#include <process.h>
int wait(int *statloc);
```

Description

Waits for one or more child processes to terminate.

The *wait* function waits for one or more child processes to terminate. The child processes must be those created by the calling program; *wait* cannot wait for grandchildren (processes spawned by child processes). If *statloc* is not NULL, it points to location where *wait* will store the termination status.

If the child process terminated normally (by calling *exit*, or returning from *main*), the termination status word is defined as follows:

Bits 0-7	Zero.
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to *exit*, or is returned from *main*. If the child process simply exited from *main* without returning a value, this value will be unpredictable. If the child process terminated abnormally, the termination status word is defined as follows:
Bits 0-7	Termination information about the child: 1 Critical error abort. 2 Execution fault, protection exception. 3 External termination signal.
Bits 8-15	Zero.

Return Value

When *wait* returns after a normal child process termination it returns the process ID of the child.

When *wait* returns after an abnormal child termination it returns -1 to the parent and sets *errno* to EINTR.

If *wait* returns without a child process completion it returns a -1 value and sets errno to

ECHILD	No child process exists

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			+			+

wcstombs

Example Portability

Syntax

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description

Converts a **wchar_t** array into a multibyte string.

wcstombs converts the type **wchar_t** elements contained in *pwcs* into a multibyte character string *s*. The process terminates if either a null character or an invalid multibyte character is encountered.

No more than *n* bytes are modified. If *n* number of bytes are processed before a null character is reached, the array *s* is not null terminated.

The behavior of *wcstombs* is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If an invalid multibyte character is encountered, *wcstombs* returns (size_t) -1. Otherwise, the function returns the number of bytes modified, not including the terminating code, if any.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

wctomb

Example Portability

Syntax

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

Description

Converts **wchar_t** code to a multibyte character.

If *s* is not null, *wctomb* determines the number of bytes needed to represent the multibyte character corresponding to *wc* (including any change in shift state). The multibyte character is stored in *s*. At most MB_CUR_MAX characters are stored. If the value of *wc* is zero, *wctomb* is left in the initial state.

The behavior of *wctomb* is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If *s* is a NULL pointer, *wctomb* returns a nonzero value if multibyte character encodings do have state-dependent encodings, and a zero value if they do not.

If *s* is not a NULL pointer, *wctomb* returns -1 if the *wc* value does not represent a valid multibyte character. Otherwise, *wctomb* returns the number of bytes that are contained in the multibyte character corresponding to *wc*. In no case will the return value be greater than the value of MB_CUR_MAX macro.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

wherex

See also Example Portability

Syntax

```
#include <conio.h>
int wherex(void);
```

Description

Gives horizontal cursor position within window.

Note: Do not use this function for Win32s or Win32 GUI applications.

wherex returns the x-coordinate of the current cursor position (within the current text window).

Return Value

wherex returns an integer in the range 1 to the number of columns in the current video mode.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

wherey

See also Example Portability

Syntax

```
#include <conio.h>
int wherey(void);
```

Description

Gives vertical cursor position within window.

Note: Do not use this function for Win32s or Win32 GUI applications.

wherey returns the y-coordinate of the current cursor position (within the current text window).

Return Value

wherey returns an integer in the range 1 to the number of rows in the current video mode.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

window

See also Example Portability

Syntax

```
#include <conio.h>
void window(int left, int top, int right, int bottom);
```

Description

Defines active text mode window.

Note: Do not use this function for Win32s or Win32 GUI applications.

window defines a text window onscreen. If the coordinates are in any way invalid, the call to *window* is ignored.

left and *top* are the screen coordinates of the upper left corner of the window.

right and *bottom* are the screen coordinates of the lower right corner.

The minimum size of the text window is one column by one line. The default window is full screen, with the coordinates:

```
  1,1,C,R
```

where C is the number of columns in the current video mode, and R is the number of rows.

Return Value

None.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+			+			+

See Also

lseek

_rtl_read

write

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

write

See also Example Portability

Syntax

```
#include <io.h>
int write(int handle, void *buf, unsigned len);
```

Description

Writes to a file.

write writes a buffer of data to the file or device named by the given *handle*. *handle* is a file handle obtained from a *creat, open, dup,* or *dup2* call.

This function attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle*. Except when *write* is used to write to a text file, the number of bytes written to the file will be no more than the number requested. The maximum number of bytes that *write* can write is UINT_MAX -1, because UINT_MAX is the same as -1, which is the error return indicator for *write*. On text files, when *write* sees a linefeed (LF) character, it outputs a CR/LF pair. UINT_MAX is defined in limits.h.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disks or disk files, writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device. For files opened with the O_APPEND option, the file pointer is positioned to EOF by *write* before writing the data.

Return Value

write returns the number of bytes written. A *write* to a text file does not count generated carriage returns. In case of error, *write* returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

/* abs example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  int number = -1234;

  printf("number: %d  absolute value: %d\n", number, abs(number));
  return 0;
}
```

/* cabs example */

```
#include <stdio.h>
#include <math.h>

#ifdef __cplusplus
  #include <complex.h>
#endif

#ifdef __cplusplus /* if C++, use class complex */

  void print_abs(void)
  {
    complex z(1.0, 2.0);
    double  absval;

    absval = abs(z);
    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      real(z), imag(z), absval);
  }

#else  /* below function is for C (and not C++) */

  void print_abs(void)
  {
    struct complex z;
    double absval;

    z.x = 2.0;
    z.y = 1.0;
    absval = cabs(z);

    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      z.x, z.y, absval);
  }

#endif

int main(void)
{
  print_abs();
  return 0;
}
```

/* fabs example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   float  number = -1234.0;

   printf("number: %f  absolute value: %f\n", number, fabs(number));
   return 0;
}
```

/* labs example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   long result;
   long x = -12345678L;

   result= labs(x);
   printf("number: %ld abs value: %ld\n", x, result);

   return 0;
}
```

/* acos example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  double result;
  double x = 0.5;

  result = acos(x);
  printf("The arc cosine of %lf is %lf\n", x, result);
  return 0;
}
```

/* asin example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 0.5;
   result = asin(x);
   printf("The arc sin of %lf is %lf\n", x, result);
  return(0);
}
```

/* atan example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 0.5;

   result = atan(x);
   printf("The arc tangent of %lf is %lf\n", x, result);
  return(0);
}
```

/* atan2 example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 90.0, y = 45.0;

   result = atan2(y, x);
   printf("The arc tangent ratio of %lf is %lf\n", (y / x), result);
   return 0;
}
```

/* alloca example */

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

void test(int a)
{
   char *newstack;
   int len = a;
   char dummy[1];

  dummy[0] = 0;          /* force good stack frame */
   printf("SP before calling alloca(0x%X) = 0x%X\n",len,_SP);
   newstack = (char *) alloca(len);
  printf("SP after calling alloca = 0x%X\n",_SP);
   if (newstack)
      printf("Alloca(0x%X) returned %p\n",len,newstack);
   else
      printf("Alloca(0x%X) failed\n",len);
}
void main()
{
   test(256);
   test(16384);
}
```

/* asctime example */

```
#include <string.h>
#include <time.h>
#include <stdio.h>

int main(void)
{
  struct tm t;
  char str[80];

  /* sample loading of tm structure  */

  t.tm_sec    = 1;  /* Seconds */
  t.tm_min    = 30; /* Minutes */
  t.tm_hour   = 9;  /* Hour */
  t.tm_mday   = 22; /* Day of the Month  */
  t.tm_mon    = 11; /* Month */
  t.tm_year   = 56; /* Year - does not include century */
  t.tm_wday   = 4;  /* Day of the week  */
  t.tm_yday   = 0;  /* Does not show in asctime */
  t.tm_isdst  = 0;  /* Is Daylight SavTime; does not show in asctime */

  /* converts structure to null terminated string */

  strcpy(str, asctime(&t));
  printf("%s\n", str);

  return 0;
}
```

/* ctime example */

```
#include <stdio.h>
#include <time.h>

int main(void)
{
  time_t t;

  time(&t);
  printf("Today's date and time: %s\n", ctime(&t));
  return 0;
}
```

/* _beginthread example */

```
#include <stdio.h>
#include <errno.h>
#include <stddef.h>     /* _threadid variable */
#include <process.h>    /* _beginthread, _endthread */
#include <time.h>       /* time, _ctime */

void thread_code(void *threadno)
{
    time_t t;

    time(&t);
    printf("Executing thread number %d, ID = %d, time = %s\n",
        (int)threadno, _threadid, ctime(&t));
    _endthread();
}

void start_thread(int i)
{
    int thread_id;

#if     defined(__WIN32__)
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == (unsigned
  long)-1)
#else
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == -1)
#endif
    {
        printf("Unable to create thread %d, errno = %d\n",i,errno);
        return;
    }
    printf("Created thread %d, ID = %ld\n",i,thread_id);
}

int main(void)
{
    int i;

    for (i = 1; i < 20; i++)
        start_thread(i);
    printf("Hit ENTER to exit main thread.\n");
    getchar();
    return 0;
}
```

/* beginthreadNT example */

```
#include        <windows.h>
#include        <process.h>
#include        <stdio.h>
#include        <conio.h>


/*      This function acts as the 'main' function for each new thread.
static  void    threadMain(void *arg) */
{
        printf("Thread %2d has an ID of %u\n", (int)arg,
  GetCurrentThreadId());

        _endthread();
}


int     main(void)
{
        #define NTHREADS        25

        HANDLE  hThreads[NTHREADS];
        int     i;

        //      Create NTHREADS inheritable threads that are initially
        //      suspended and that will run starting at threadMain().
        //      at threadMain().
        for     (i = 0; i < NTHREADS; i++)
        {
                SECURITY_ATTRIBUTES     sa =
                {
                        sizeof(SECURITY_ATTRIBUTES),    // structure size
                        0,                              // No security
  descriptor
                        TRUE,                           // Thread handle is
  inheritable
                };

                DWORD   threadId;

                hThreads[i] = (HANDLE)_beginthreadNT(
                                        threadMain,             // Thread
  starting address
                                        4096,                   // Thread
  stack size
                                        (void *)i,              // Thread
  start argument
                                        &sa,                    // Thread
  security
                                        CREATE_SUSPENDED,       // Create
  in suspended state
                                        &threadId);             // Thread
  ID.

                if      (hThreads[i] == INVALID_HANDLE_VALUE)
                {
```

```
                        MessageBox(0, "Thread Creation Failed", "Error",
  MB_OK);
                        return  1;
                }

                printf("Created thread %2d with an ID of %u\n", i,
  threadId);
        }

        printf("\nPress a key to thaw all threads\n\n");
        getch();

        //      Resume the suspended threads.
        for     (i = 0; i < NTHREADS; i++)
                ResumeThread(hThreads[i]);

        //      Wait for all threads to finish execution.
        WaitForMultipleObjects(NTHREADS,        // Number of objects to
  wait for
                               hThreads,        // The objects to wait for
                               TRUE,            // Wait for all objects
                               INFINITE);       // No timeout

        //      Close all of the thread handles.
        for     (i = 0; i < NTHREADS; i++)
                CloseHandle(hThreads[i]);

        return  0;
}
```

/* biosequip example */

```
#include <bios.h>
#include <stdio.h>

#define CO_PROCESSOR_MASK 0x0002

int main(void)
{
  int equip_check;

   /* get the current equipment configuration */
   equip_check = biosequip();

   /* check to see if there is a coprocessor installed */
   if (equip_check & CO_PROCESSOR_MASK)
      printf("There is a math coprocessor installed.\n");
   else
      printf("No math coprocessor installed.\n");
   return 0;
}
```

/* _bios_equiplist example */

```
#include <stdio.h>
#include <bios.h>

#define CO_PROCESSOR_MASK 0x0002

int main(void)
{
  unsigned equip_check;

  /* get the current equipment configuration */
  equip_check = _bios_equiplist();

  /* check to see if there is a coprocessor installed */
  if (equip_check & CO_PROCESSOR_MASK)
    printf("There is a math coprocessor installed.\n");
   else
    printf("No math coprocessor installed.\n");
  return 0;
}
```

/* biosmemory example */

```
#include <stdio.h>
#include <bios.h>

int main(void)
{
   int memory_size;

   memory_size = biosmemory();  /* returns value up to 640K */
   printf("RAM size = %dK\n",memory_size);
   return 0;
}
```

/* _bios_memsize example */

```
#include <stdio.h>
#include <bios.h>

int main(void)
{
  unsigned   memory_size;

  memory_size = _bios_memsize();   /* returns value up to 640K */

  printf("RAM size = %dK\n", memory_size);

  return 0;
}
```

/* biostime example */

```
#include <stdio.h>
#include <bios.h>
#include <time.h>
#include <conio.h>

int main(void)
{
  long bios_time;
   clrscr();
   printf("The number of clock ticks since midnight is:\n");
   printf("The number of seconds since midnight is:\n");
   printf("The number of minutes since midnight is:\n");
   printf("The number of hours since midnight is:\n");
   printf("\nPress any key to stop:");
   while(!kbhit())
   {
      bios_time = biostime(0, 0L);

      gotoxy(50, 1);
      printf("%lu", bios_time);

      gotoxy(50, 2);
      printf("%.4f", bios_time / _BIOS_CLK_TCK);

      gotoxy(50, 3);
      printf("%.4f", bios_time / _BIOS_CLK_TCK / 60);

      gotoxy(50, 4);
      printf("%.4f", bios_time / _BIOS_CLK_TCK / 3600);
   }
   return 0;
}
```

/* _bios_timeofday example */

```
#include <bios.h>
#include <time.h>
#include <conio.h>
#include <stdio.h>

int main(void)
{
  long bios_time;
  clrscr();
  printf("The number of clock ticks since midnight is:\n");
  printf("The number of seconds since midnight is:\n");
  printf("The number of minutes since midnight is:\n");
  printf("The number of hours since midnight is:\n");
  printf("\nPress any key to stop:");
  while(!kbhit())
  {
    _bios_timeofday(_TIME_GETCLOCK, &bios_time);
    gotoxy(50, 1);
    printf("%lu", bios_time);
    gotoxy(50, 2);
    printf("%.4f", bios_time / CLK_TCK);
    gotoxy(50, 3);
    printf("%.4f", bios_time / CLK_TCK / 60);
    gotoxy(50, 4);
    printf("%.4f", bios_time / CLK_TCK / 3600);
  }
  return 0;
}
```

/* bsearch example */

```
#include <stdlib.h>
#include <stdio.h>

typedef int (*fptr)(const void*, const void*);

#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))

int numarray[] = {123, 145, 512, 627, 800, 933};

int numeric (const int *p1, const int *p2)
{
   return(*p1 - *p2);
}

#pragma argsused
int lookup(int key)
{
   int  *itemptr;

   /* The cast of (int(*)(const void *,const void*))
      is needed to avoid a type mismatch error at
      compile time */
   itemptr = (int *) bsearch (&key, numarray, NELEMS(numarray),
      sizeof(int), (fptr)numeric);
   return (itemptr != NULL);
}

int main(void)
{
   if (lookup(512))
      printf("512 is in the table.\n");
   else
      printf("512 isn't in the table.\n");

   return 0;
}
```

/* lfind example */

```
#include <stdio.h>
#include <stdlib.h>

int compare(int *x, int *y)
{
   return( *x - *y );
}

int main(void)
{
   int array[5] = {35, 87, 46, 99, 12};
   size_t nelem = 5;
   int key;
   int *result;

   key = 99;
   result = (int *) lfind(&key, array, &nelem,
      sizeof(int), (int(*)(const void *,const void *))compare);
   if (result)
      printf("Number %d found\n",key);
   else
      printf("Number %d not found\n",key);

   return 0;
}
```

/* lsearch example */

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>     /* for strcmp declaration */

/* initialize number of colors */
char *colors[10] = { "Red", "Blue", "Green" };
int ncolors = 3;

int colorscmp(char **arg1, char **arg2)
{
   return(strcmp(*arg1, *arg2));
}

int addelem(char **key)
{
   int oldn = ncolors;
   lsearch(key, colors, (size_t *)&ncolors, sizeof(char *),
     (int(*)(const void *,const void *))colorscmp);
   return(ncolors == oldn);
}

int main(void)
{
   int i;
   char *key = "Purple";

   if (addelem(&key))
      printf("%s already in colors table\n", key);
   else
   {
      printf("%s added to colors table\n", key);
   }

   printf("The colors:\n");
   for (i = 0; i < ncolors; i++)
      printf("%s\n", colors[i]);
   return 0;
}
```

/* qsort example */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int sort_function( const void *a, const void *b);
char list[5][4] = { "cat", "car", "cab", "cap", "can" };

int main(void)
{
   int  x;

   qsort((void *)list, 5, sizeof(list[0]), sort_function);
   for (x = 0; x < 5; x++)
      printf("%s\n", list[x]);
   return 0;
}

int sort_function( const void *a, const void *b)
{
   return( strcmp((char *)a,(char *)b) );
}
```

/* _rtl_chmod example */

```
#include <errno.h>
#include <stdio.h>
#include <dos.h>
#include <io.h>

int get_file_attrib(char *filename);

int main(void)
{
  char filename[128];
  int attrib;
  printf("Enter a filename:");
  scanf("%s", filename);
  attrib = get_file_attrib(filename);
  if (attrib == -1)
     switch(errno)
     {
        case ENOENT : printf("Path or file not found.\n");
                      break;
        case EACCES : printf("Permission denied.\n");
                      break;
        default:      printf("Error number: %d", errno);
                      break;
     }
  else
     {
        if (attrib & FA_RDONLY)
           printf("%s is read-only.\n", filename);

        if (attrib & FA_HIDDEN)
           printf("%s is hidden.\n", filename);

        if (attrib & FA_SYSTEM)
           printf("%s is a system file.\n", filename);

        if (attrib & FA_DIREC)
           printf("%s is a directory.\n", filename);

        if (attrib & FA_ARCH)
           printf("%s is an archive file.\n", filename);
     }
  return 0;
}

/* returns the attributes of a DOS file */
int get_file_attrib(char *filename)
{
   return(_rtl_chmod(filename, 0));
}
```

/* _dos_getfileattr example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
  char filename[128];
  unsigned attrib;
  printf("Enter a file name:");
  scanf("%s", filename);
  if (_dos_getfileattr(filename,&attrib) != 0)
  {
    perror("Unable to obtain file attributes");
    return 1;
  }
  if (attrib & _A_RDONLY)
    printf("%s is read-only.\n", filename);

  if (attrib & _A_HIDDEN)
    printf("%s is hidden.\n", filename);

  if (attrib & _A_SYSTEM)
    printf("%s is a system file.\n", filename);

  if (attrib & _A_VOLID)
    printf("%s is a volume label.\n", filename);

  if (attrib & _A_SUBDIR)
    printf("%s is a directory.\n", filename);

  if (attrib & _A_ARCH)
    printf("%s is an archive file.\n", filename);
  return 0;
}
```

/* _dos_setfileattr example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
  char filename[128];
  unsigned attrib;
  printf("Enter a file name:");
  scanf("%s", filename);
  if (_dos_getfileattr(filename,&attrib) != 0)
  {
    perror("Unable to obtain file attributes");
    return 1;
  }
  if (attrib & _A_RDONLY)
  {
    printf("%s currently read-only, making it read-write.\n", filename);
    attrib &= ~_A_RDONLY;
  }
  else
  {
    printf("%s currently read-write, making it read-only.\n", filename);
    attrib |= _A_RDONLY;
  }
  if (_dos_setfileattr(filename,attrib) != 0)
    perror("Unable to set file attributes");
  return 0;
}
```

/* close example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

main()
{
  int handle;
  char buf[11] = "0123456789";

  /* create a file containing 10 bytes */
  handle = open("NEW.FIL", O_CREAT);
  if (handle > -1)
  {
    write(handle, buf, strlen(buf));

    close(handle);                    /* close the file */
  }
  else
  {
    printf("Error opening file\n");
  }
  return 0;
}
```

/* _rtl_close example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char msg[] = "Hello world";

   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

/* _dos_close example */

```
#include <dos.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
  unsigned count;
  int handle;
  char buf[11] = "0123456789";

  /* create a file containing 10 bytes */
  if (_dos_creat("DUMMY.FIL", _A_NORMAL, &handle) != 0)
  {
    perror("Unable to create DUMMY.FIL");
    return 1;
  }
  if (_dos_write(handle, buf, strlen(buf), &count) != 0)
  {
    perror("Unable to write to DUMMY.FIL");
    return 1;
  }
  /* close the file */
  _dos_close(handle);
  return 0;
}
```

/* cos example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 0.5;

   result = cos(x);
   printf("The cosine of %lf is %lf\n", x, result);
   return 0;
}
```

/* sin example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result, x = 0.5;

   result = sin(x);
   printf("The sin of %lf is %lf\n", x, result);
   return 0;
}
```

/* tan example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result, x;

   x = 0.5;
   result = tan(x);
   printf("The tan of %lf is %lf\n", x, result);
   return 0;
}
```

/* cosh example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 0.5;

   result = cosh(x);
   printf("The hyperbolic cosine of %lf is %lf\n", x, result);
   return 0;
}
```

/* sinh example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result, x = 0.5;

   result = sinh(x);
   printf("The hyperbolic sin of %lf is %lf\n", x, result);
   return 0;
}
```

/* tanh example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result, x;

   x = 0.5;
   result = tanh(x);
   printf("The hyperbolic tangent of %lf is %lf\n", x, result);
   return 0;
}
```

/* creat example */

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char buf[11] = "0123456789";

   /* change the default file mode from text to binary */
   _fmode = O_BINARY;

   /* create a binary file for reading and writing */
   handle = creat("DUMMY.FIL", S_IREAD |S_IWRITE);

   /* write 10 bytes to the file */
   write(handle, buf, strlen(buf));

   /* close the file */
   close(handle);
   return 0;
}
```

/* _rtl_creat example */

```
#include <dos.h>
#include <string.h>
#include <stdio.h>
#include <io.h>

int main() {
  unsigned count;
  int handle;
  char buf[11] = "0123456789";

  /* Create a 10-byte file using _dos_creat. */
  if (_dos_creat("DUMMY.FIL", _A_NORMAL, &handle) != 0) {
    perror("Unable to _dos_creat DUMMY.FIL");
    return 1;
  }
  if (_dos_write(handle, buf, strlen(buf), &count) != 0) {
    perror("Unable to _dos_write to DUMMY.FIL");
    return 1;
  }
  _dos_close(handle);

  /* Create another 10-byte file using _rtl_creat. */
  if ((handle = _rtl_creat("DUMMY2.FIL", 0)) < 0) {
    perror("Unable to _rtl_create DUMMY2.FIL");
    return 1;
  }
  if (_rtl_write(handle, buf, strlen(buf)) < 0) {
    perror("Unable to _rtl_write to DUMMY2.FIL");
    return 1;
  }
  _rtl_close(handle);
  return 0;
}
```

/* _dos_creat example */

```
#include <dos.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
  unsigned count;
  int handle;
  char buf[11] = "0123456789";

  /* create a file containing 10 bytes */
  if (_dos_creat("DUMMY.FIL", _A_NORMAL, &handle) != 0)
  {
    perror("Unable to create DUMMY.FIL");
    return 1;
  }
  if (_dos_write(handle, buf, strlen(buf), &count) != 0)
  {
    perror("Unable to write to DUMMY.FIL");
    return 1;
  }
  /* close the file */
  _dos_close(handle);
  return 0;
}
```

/* _dos_creatnew example */

```
#include <dos.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
  unsigned count;
  int handle;
  char buf[11] = "0123456789";

  /* create a file containing 10 bytes */
  if (_dos_creatnew("DUMMY.FIL", _A_NORMAL, &handle) != 0)
  {
    perror("Unable to create DUMMY.FIL");
    return 1;
  }
  if (_dos_write(handle, buf, strlen(buf), &count) != 0)
  {
    perror("Unable to write to DUMMY.FIL");
    return 1;
  }
  /* close the file */
  _dos_close(handle);
  return 0;
}
```

/* creatnew example */

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <dos.h>
#include <io.h>

int main(void)
{
   int handle;
   char buf[11] = "0123456789";

   /* attempt to create a file that doesn't already exist */
   handle = creatnew("DUMMY.FIL", 0);

  if (handle == -1)
      printf("DUMMY.FIL already exists.\n");
   else
   {
      printf("DUMMY.FIL successfully created.\n");
      write(handle, buf, strlen(buf));
      close(handle);
   }
   return 0;
}
```

/* disable example */

```
/* * * * * * * * * *
NOTE: This is an interrupt service routine. You cannot compile this program
  with Test Stack Overflow turned on and get an executable file that
  operates correctly.
 * * * * * * * * * * */

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt (*oldhandler)(__CPPARGS);

int count=0;

void interrupt handler(__CPPARGS) /* if C++, need the the ellipsis */
{
/* disable interrupts during the handling of the interrupt */
   disable();
/* increase the global counter */
   count++;
/* reenable interrupts at the end of the handler */
  enable();
/* call the old routine */
   oldhandler();
}

int main(void)
{
/* save the old interrupt vector */
   oldhandler = getvect(INTR);

/* install the new interrupt handler */
   setvect(INTR, handler);

/* loop until the counter exceeds 20 */
   while (count < 20)
      printf("count is %d\n",count);

/* reset the old interrupt handler */
   setvect(INTR, oldhandler);

  return 0;
}
```

/* _disable example */

```
/* * * * * * * * * *
NOTE: This is an interrupt service routine. You cannot compile this program
  with Test Stack Overflow turned on and get an executable file that
  operates correctly.
 * * * * * * * * * * */

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt (*oldhandler)(__CPPARGS);

int count=0;

void interrupt handler(__CPPARGS) /* if C++, need the the ellipsis */
{
/* disable interrupts during the handling of the interrupt */
   _disable();
/* increase the global counter */
   count++;
/* reenable interrupts at the end of the handler */
   enable();
/* call the old routine */
   oldhandler();
}

int main(void)
{
/* save the old interrupt vector */
  oldhandler = _dos_getvect(INTR);

/* install the new interrupt handler */
   _dos_setvect(INTR, handler);

/* loop until the counter exceeds 20 */
   while (count < 20)
      printf("count is %d\n",count);

/* reset the old interrupt handler */
   _dos_setvect(INTR, oldhandler);

   return 0;
}
```

/* enable example */

```
/* * * * * * * * * *
NOTE: This is an interrupt service routine. You cannot compile this program
  with Test Stack Overflow turned on and get an executable file that
  operates correctly.
 * * * * * * * * * * */

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt (*oldhandler)(__CPPARGS);

int count=0;

void interrupt handler(__CPPARGS) /* if C++, need the the ellipsis */
{
/* disable interrupts during the handling of the interrupt */
   disable();
/* increase the global counter */
   count++;
/* reenable interrupts at the end of the handler */
   enable();
/* call the old routine */
   oldhandler();
}

int main(void)
{
/* save the old interrupt vector */
   oldhandler = getvect(INTR);

/* install the new interrupt handler */
   setvect(INTR, handler);

/* loop until the counter exceeds 20 */
   while (count < 20)
      printf("count is %d\n",count);

/* reset the old interrupt handler */
   setvect(INTR, oldhandler);

   return 0;
}
```

/* _enable example */

```
/* * * * * * * * * *
NOTE: This is an interrupt service routine. You cannot compile this program
  with Test Stack Overflow turned on and get an executable file that
  operates correctly.
 * * * * * * * * * * */

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt (*oldhandler)(__CPPARGS);

int count=0;

void interrupt handler(__CPPARGS) /* if C++, need the the ellipsis */
{
/* disable interrupts during the handling of the interrupt */
   disable();
/* increase the global counter */
   count++;
/* reenable interrupts at the end of the handler */
   _enable();
/* call the old routine */
   oldhandler();
}

int main(void)
{
/* save the old interrupt vector */
   oldhandler = _dos_getvect(INTR);

/* install the new interrupt handler */
   _dos_setvect(INTR, handler);

/* loop until the counter exceeds 20 */
   while (count < 20)
      printf("count is %d\n",count);

/* reset the old interrupt handler */
   _dos_setvect(INTR, oldhandler);

   return 0;
}
```

/* div example */

```
/* div example */

#include <stdlib.h>
#include <stdio.h>

div_t x;

int main(void)
{
   x = div(10,3);
   printf("10 div 3 = %d remainder %d\n",
   x.quot, x.rem);

   return 0;
}
```

/* ldiv example */

```
/* ldiv example */

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   ldiv_t lx;

   lx = ldiv(100000L, 30000L);
  printf("100000 div 30000 = %ld remainder %ld\n", lx.quot, lx.rem);
   return 0;
}
```

/* dup example */

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <io.h>

void flush(FILE *stream);

int main(void)
{
  FILE *fp;
  char msg[] = "This is a test";

  /* create a file */
  fp = fopen("DUMMY.FIL", "w");

  /* write some data to the file */
  fwrite(msg, strlen(msg), 1, fp);

  clrscr();
  printf("Press any key to flush DUMMY.FIL:");
  getch();

  /* flush the data to DUMMY.FIL without closing it */
  flush(fp);

  printf("\nFile was flushed, Press any key to quit:");
  getch();
  return 0;
}

void flush(FILE *stream)
{
  int duphandle;

  /* flush TC's internal buffer */
  fflush(stream);

  /* make a duplicate file handle */
  duphandle = dup(fileno(stream));

  /* close the duplicate handle to flush the DOS buffer */
  close(duphandle);
}
```

/* dup2 example */

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   #define STDOUT 1

   int nul, oldstdout;
   char msg[] = "This is a test";

   /* create a file */
   nul = open("DUMMY.FIL", O_CREAT | O_RDWR,
      S_IREAD | S_IWRITE);

   /* create a duplicate handle for standard output */
   oldstdout = dup(STDOUT);
   /*
      redirect standard output to DUMMY.FIL
      by duplicating the file handle onto
      the file handle for standard output.
   */
   dup2(nul, STDOUT);

   /* close the handle for DUMMY.FIL */
   close(nul);

   /* will be redirected into DUMMY.FIL */
   write(STDOUT, msg, strlen(msg));

   /* restore original standard output handle */
   dup2(oldstdout, STDOUT);

   /* close duplicate handle for STDOUT */
   close(oldstdout);

   return 0;
}
```

/* ecvt example */

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char *string;
   double value;
   int dec, sign;
   int ndig = 10;

   clrscr();
   value = 9.876;
   string = ecvt(value, ndig, &dec, &sign);
   printf("string = %s      dec = %d sign = %d\n", string, dec, sign);

   value = -123.45;
   ndig= 15;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s dec = %d sign = %d\n", string, dec, sign);

   value = 0.6789e5; /* scientific notation */
   ndig = 5;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s         dec = %d sign = %d\n", string, dec, sign);

   return 0;
}
```

/* fcvt example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  char *str;
  double num;
  int dec, sign, ndig = 5;

  /* a regular number */
  num = 9.876;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);

  /* a negative number */
  num = -123.45;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);

  /* scientific notation */
  num = 0.678e5;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s  decimal place= %d  sign = %d\n", str, dec, sign);
  return 0;
}
```

/* execl example */

```
/* execl() example */
#include <stdio.h>
#include <process.h>

int main(int argc, char *argv[])
{
  int loop;

  printf("%s running...\n\n", argv[0]);

  if (argc == 1) {    /* check for only one command-line parameter */
    printf("%s calling itself again...\n", argv[0]);
    execl(argv[0], argv[0], "ONE", "TWO", "THREE", NULL);
    perror("EXEC:");
    exit(1);
  }

  printf("%s called with arguments:\n", argv[0]);

  for (loop = 1; loop <= argc; loop++)
    puts(argv[loop]);       /* Display all command-line parameters */
  return 0;
}
```

/* execlp example */

```
/* execlp example */
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main( int argc, char *argv[])
{
   int i;

   printf("Command line arguments:\n");

   for (i=0; i < argc; ++i)
      printf("[%2d] %s\n", i, argv[i]);

   printf("About to exec child with arg1 arg2 ...\n");
   execlp("CHILD.EXE", "CHILD.EXE", "arg1", "arg2", NULL);

   perror("exec error");
   exit(1);

   return 0;
}
```

/* execle example */

```
#include <process.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
  int loop;
  char *new_env[] = { "TESTING", NULL };
  printf("%s running...\n\n", argv[0]);
  if (argc == 1) {   /* check for only one command-line parameter */
    printf("%s calling itself again...\n", argv[0]);
    execle(argv[0], argv[0], "ONE", "TWO", "THREE", NULL, new_env);
    perror("EXEC:");
    exit(1);
  }
  printf("%s called with arguments:\n", argv[0]);
  for (loop = 1; loop <= argc; loop++)
    puts(argv[loop]);  /* display all command-line parameters */

  /* display the first environment parameter */
  printf("value of env[0]: %s\n",env[0]);
  return 0;
}
```

/* execlpe example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[], char **envp )
{
  int i;

  printf("Command line arguments:\n");
  for (i=0; i < argc; ++i)
    printf("[%2d] : %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  execlpe("CHILD.EXE", "CHILD.EXE", "arg1", "arg2", NULL, envp);

  perror("exec error");
  exit(1);
  return 0;
}
```

/* execv example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
  int i;

  printf("Command line arguments:\n");
  for (i=0; i < argc; i++)
    printf("[%2d] : %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  execv("CHILD.EXE", argv);

  perror("exec error");
  exit(1);
  return 0;
}
```

/* execve example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[], char **envp)
{
  int i;

  printf("Command line arguments:\n");
  for (i=0; i < argc; ++i)
    printf("[%2d] : %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  execve("CHILD.EXE", argv, envp);

  perror("exec error");
  exit(1);
  return 0;
}
```

/* execvp example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
  int i;

  printf("Command line arguments:\n");
  for (i=0; i < argc; ++i)
    printf("[%2d] : %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  execvp("CHILD.EXE", argv);

  perror("exec error");
  exit(1);
  return 0;
}
```

/* execvpe example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[], char **envp)
{
  int i;

  printf("Command line arguments:\n");
  for (i=0; i < argc; ++i)
    printf("[%2d] : %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  execvpe("CHILD.EXE", argv, envp);

  perror("exec error");
  exit(1);
  return 0;
}
```

/* _exit example */

```
#include   <stdlib.h>
#include   <stdio.h>

void done(void);

int main(void)
{

   atexit(done);
   _exit(0);
   return 0;
}

void done()
{
   printf("hello\n");
}
```

/* _c_exit example */

```
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <dos.h>

main()
{
  int fd;
  char c;

  if ((fd = open("_c_exit.c",O_RDONLY)) < 0)
  {
    printf("Unable to open _c_exit.c for reading\n");
    return 1;
  }
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d before _c_exit\n",fd);
  else
    printf("Successfully read from open file handle %d before _c_exit\
 n",fd);
  printf("Interrupt zero vector before _c_exit = %Fp\n",_dos_getvect(0));
  _c_exit();
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d after _c_exit\n",fd);
  else
    printf("Successfully read from open file handle %d after _c_exit\
 n",fd);
  printf("Interrupt zero vector after _c_exit = %Fp\n",_dos_getvect(0));
 return 0;
}
```

/* exit */

```
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int status;

   printf("Enter either 1 or 2\n");
   status = getch();
   /* Sets DOS errorlevel  */
   exit(status - '0');

/* Note: this line is never reached */
   return 0;
}
```

```
/* _cexit example */
#include <windows.h>
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

void exit_func(void)
{
  printf("Exit function called\n\n");
  printf("Close Window to return to program... It will beep if able to read
  from file");
}

int main(void)
{
  int fd;
  char c;

  if ((fd = open("_cexit.c",O_RDONLY)) < 0)
  {
  printf("Unable to open _cexit.c for reading\n");
  return 1;
  }
  atexit(exit_func);
  if (read(fd,&c,1) != 1)
  printf("Unable to read from open file handle %d before _cexit\n",fd);
  else
  printf("Successfully read from open file handle %d before _cexit\n",fd);
  _cexit();
  if (read(fd,&c,1) == 1)
    MessageBeep(0);
  return 0;
}
```

/* farfree example */

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
   char far *fptr;
   char *str = "Hello";

   /* allocate memory for the far pointer */
   fptr = (char far *) farcalloc(10, sizeof(char));

   /* copy "Hello" into allocated memory */
/*
Note: movedata is used because you might be in a small data model, in which
  case a normal string copy routine can't be used since it assumes the
  pointer size is near.
 */
   movedata(FP_SEG(str), FP_OFF(str),
            FP_SEG(fptr), FP_OFF(fptr),
            strlen(str));

   /* display string (note the F modifier) */
   printf("Far string is: %Fs\n", fptr);

   /* free the memory */
   farfree(fptr);

   return 0;
}
```

/* free example */

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>

int main(void)
{
   char *str;

   /* allocate memory for string */
   str = (char *) malloc(10);

   /* copy "Hello" to string */
   strcpy(str, "Hello");

   /* display string */
   printf("String is %s\n", str);

   /* free memory */
   free(str);

   return 0;
}
```

/* fgetc example */

```
#include <string.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char ch;

   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");

   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);

   /* seek to the beginning of the file */
   fseek(stream, 0, SEEK_SET);

   do
   {
      /* read a char from the file */
      ch = fgetc(stream);

      /* display the character */
      putch(ch);
   } while (ch != EOF);

   fclose(stream);
   return 0;
}
```

/* fputc example */

```
#include <stdio.h>

int main(void)
{
   char msg[] = "Hello world";
   int i = 0;

   while (msg[i])
   {
      fputc(msg[i], stdout);
      i++;
   }
   return 0;
}
```

/* fgetchar example */

```
#include <stdio.h>

int main(void)
{
   char ch;

   /* prompt the user for input */
   printf("Enter a character followed by <Enter>: ");

   /* read the character from stdin */
   ch = fgetchar();

   /* display what was read */
   printf("The character read is: '%c'\n", ch);
   return 0;
}
```

/* fputchar example */

```
#include <stdio.h>

int main(void)
{
   char msg[] = "This is a test";
   int i = 0;

   while (msg[i])
   {
      fputchar(msg[i]);
      i++;
   }
   return 0;
}
```

/* fgets example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char msg[20];

   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");

   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);

   /* seek to the start of the file */
   fseek(stream, 0, SEEK_SET);

   /* read a string from the file */
   fgets(msg, strlen(string)+1, stream);

   /* display the string */
   printf("%s", msg);

   fclose(stream);
   return 0;
}
```

/* fputs example */

```
#include <stdio.h>

int main(void)
{
   /* write a string to standard output */
   fputs("Hello world\n", stdout);

   return 0;
}
```

/* _dos_findfirst and _dos_findnext example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
  struct find_t ffblk;
  int done;
  printf("Directory listing of *.*\n");
  done = _dos_findfirst("*.*",_A_NORMAL,&ffblk);
  while (!done) {
    printf("  %s\n", ffblk.name);
    done = _dos_findnext(&ffblk);
  }
  return 0;
}

/* Program output

Directory listing of *.*
  FINDFRST.C
  FINDFRST.OBJ
  FINDFRST.MAP
  FINDFRST.EXE  */
```

/* findfirst and findnext example */

```
/* findfirst and findnext example */

#include <stdio.h>
#include <dir.h>

int main(void)
{
   struct ffblk ffblk;
   int done;
   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,0);
   while (!done)
   {
      printf("  %s\n", ffblk.ff_name);
      done = findnext(&ffblk);
   }

   return 0;
}
```

```
/* _fsopen example */
#include <io.h>
#include <process.h>
#include <share.h>
#include <stdio.h>

int main(void)
{
  FILE *f;
  int status;
  f = _fsopen("c:\\autoexec.bat", "r", SH_DENYNO);
  if (f == NULL)
  {
    printf("_fsopen failed\n");
    exit(1);
  }
  status = access("c:\\autoexec.bat", 6);
  if (status == 0)
    printf("read/write access allowed\n");
  else
    printf("read/write access not allowed\n");
  fclose(f);
  return 0;
}
```

/* fdopen example */

```
#include <sys\stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   FILE *stream;

   /* open a file */
   handle = open("DUMMY.FIL", O_CREAT,
                 S_IREAD | S_IWRITE);

   /* now turn the handle into a stream */
   stream = fdopen(handle, "w");

   if (stream == NULL)
      printf("fdopen failed\n");
   else
   {
      fprintf(stream, "Hello world\n");
      fclose(stream);
   }
   return 0;
}
```

/* fopen example */

```
/* Program to create backup of the AUTOEXEC.BAT file */

#include <stdio.h>

int main(void)
{
   FILE *in, *out;

   if ((in = fopen("\\AUTOEXEC.BAT", "rt"))
       == NULL)
   {
      fprintf(stderr, "Cannot open input file.\n");
      return 1;
   }

   if ((out = fopen("\\AUTOEXEC.BAK", "wt"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }

   while (!feof(in))
      fputc(fgetc(in), out);

   fclose(in);
   fclose(out);
   return 0;
}
```

/* freopen example */

```
#include <stdio.h>

int main(void)
{
   /* redirect standard output to a file */
   if (freopen("OUTPUT.FIL", "w", stdout)
       == NULL)
      fprintf(stderr, "error redirecting stdout\n");

   /* this output will go to a file */
   printf("This will go into a file.");

   /* close the standard output stream */
   fclose(stdout);

   return 0;
}
```

/* freemem example */

```
#include <dos.h>
#include <alloc.h>
#include <stdio.h>

int main(void)
{
   unsigned int size, segp;
   int stat;

   size = 64; /* (64 x 16) = 1024 bytes */
   stat = allocmem(size, &segp);
   if (stat < 0)
      printf("Allocated memory at segment: %x\n", segp);
   else
      printf("Failed: maximum number of\
             paragraphs available is %u\n", stat);
   freemem(segp);

   return 0;
}
```

/* fstat example */

```
#include <sys\stat.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
   struct stat statbuf;
   FILE *stream;

   /* open a file for update */
   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return(1);
   }
   fprintf(stream, "This is a test");
   fflush(stream);

   /* get information about the file */
   fstat(fileno(stream), &statbuf);
   fclose(stream);

   /* display the information returned */
   if (statbuf.st_mode & S_IFCHR)
      printf("Handle refers to a device.\n");
   if (statbuf.st_mode & S_IFREG)
      printf("Handle refers to an ordinary file.\n");
   if (statbuf.st_mode & S_IREAD)
      printf("User has read permission on file.\n");
   if (statbuf.st_mode & S_IWRITE)
      printf("User has write permission on file.\n");

   printf("Drive letter of file: %c\n", 'A'+statbuf.st_dev);
   printf("Size of file in bytes: %ld\n", statbuf.st_size);
   printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
   return 0;
}
```

/* stat example */

```
#include <sys\stat.h>
#include <stdio.h>
#include <time.h>

#define FILENAME "TEST.$$$"

int main(void)
{
   struct stat statbuf;
   FILE *stream;

   /* open a file for update */
   if ((stream = fopen(FILENAME, "w+")) == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return(1);
   }

   /* get information about the file */
   stat(FILENAME, &statbuf);

   fclose(stream);

   /* display the information returned */
   if (statbuf.st_mode & S_IFCHR)
      printf("Handle refers to a device.\n");
   if (statbuf.st_mode & S_IFREG)
      printf("Handle refers to an ordinary file.\n");
   if (statbuf.st_mode & S_IREAD)
      printf("User has read permission on file.\n");
   if (statbuf.st_mode & S_IWRITE)
      printf("User has write permission on file.\n");

   printf("Drive letter of file: %c\n", 'A'+statbuf.st_dev);
   printf("Size of file in bytes: %ld\n", statbuf.st_size);
   printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
   return 0;
}
```

/* getc example */

```
#include <stdio.h>

int main(void)
{
   char ch;

   printf("Input a character:");
/* read a character from the
  standard input stream */
   ch = getc(stdin);
   printf("The character input was: '%c'\n", ch);
   return 0;
}
```

/* putc example */

```
#include <stdio.h>

int main(void)
{
   char msg[] = "Hello world\n";
   int i = 0;

   while (msg[i])
      putc(msg[i++], stdout);
   return 0;
}
```

/* getch example */

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
  int c;
  int extended = 0;
  c = getch();
  if (!c)
    extended = getch();
  if (extended)
    printf("The character is extended\n");
  else
    printf("The character isn't extended\n");

  return 0;
}
```

/* getche example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;

   printf("Input a character:");
   ch = getche();
   printf("\nYou input a '%c'\n", ch);
   return 0;
}
```

/* getchar example */

```
#include <stdio.h>

int main(void)
{
   int c;

/*
Note that getchar reads from stdin and is line buffered; this means it will
  not return until you press ENTER.
 */

   while ((c = getchar()) != '\n')
      printf("%c", c);

   return 0;
}
```

/* putchar example */

```
#include <stdio.h>

/* define some box-drawing characters */
#define LEFT_TOP  0xDA
#define RIGHT_TOP 0xBF
#define HORIZ     0xC4
#define VERT      0xB3
#define LEFT_BOT  0xC0
#define RIGHT_BOT 0xD9

int main(void)
{
   char i, j;

   /* draw the top of the box */
   putchar(LEFT_TOP);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_TOP);
   putchar('\n');

   /* draw the middle */
   for (i=0; i<4; i++)
   {
      putchar(VERT);
      for (j=0; j<10; j++)
         putchar(' ');
      putchar(VERT);
      putchar('\n');
   }

   /* draw the bottom */
   putchar(LEFT_BOT);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_BOT);
   putchar('\n');

   return 0;
}
```

/* getcwd example */

```
#include <stdio.h>
#include <dir.h>

int main(void)
{
   char buffer[MAXPATH];

   getcwd(buffer, MAXPATH);
   printf("The current directory is: %s\n", buffer);
   return 0;
}
```

```
/* _getdcwd example */
#include <direct.h>
#include <stdio.h>

char buf[65];

void main()
{
  if (_getdcwd(3, buf, sizeof(buf)) == NULL)
    perror("Unable to get current directory of drive C");
  else
    printf("Current directory of drive C is %s\n",buf);
}
```

```
/* _dos_getdate example */
#include <dos.h>
#include <stdio.h>

int main(void)
{
  struct dosdate_t d;
  _dos_getdate(&d);
  printf("The current year is: %d\n", d.year);
  printf("The current day is: %d\n", d.day);
  printf("The current month is: %d\n", d.month);
  return 0;
}
```

```
/* _dos_setdate example */
#include <dos.h>
#include <process.h>
#include <stdio.h>

int main(void)
{
  struct dosdate_t reset;
  reset.year  = 2001;
  reset.day   = 1;
  reset.month = 1;
  printf("Setting date to 1/1/2001.\n");
  _dos_setdate(&reset);
  system("date");
  return 0;
}
```

/* getdate example */

```
#include <dos.h>
#include <stdio.h>

int main(void)
{
   struct date d;

   getdate(&d);
   printf("The current year is: %d\n", d.da_year);
   printf("The current day is: %d\n", d.da_day);
   printf("The current month is: %d\n", d.da_mon);
   return 0;
}
```

```
/* setdate example */
#include <stdio.h>
#include <process.h>
#include <dos.h>

int main(void)
{
   struct date reset;
   struct date save_date;

   getdate(&save_date);
   printf("Original date:\n");
   system("date");

   reset.da_year = 2001;
   reset.da_day = 1;
   reset.da_mon = 1;
   setdate(&reset);

   printf("Date after setting:\n");
   system("date");

   setdate(&save_date);
   printf("Back to original date:\n");
   system("date");

   return 0;
}
```

/* _dos_getdiskfree example */

```
#include <stdio.h>
#include <dos.h>
#include <process.h>

int main(void)
{
  struct diskfree_t free;
  long avail;

  if (_dos_getdiskfree(0, &free) != 0) {
    printf("Error in _dos_getdiskfree() call\n");
    exit(1);
  }
  avail = (long) free.avail_clusters
        * (long) free.bytes_per_sector
        * (long) free.sectors_per_cluster;
  printf("The current drive has %ld bytes available\n", avail);
  return 0;
}
```

/* getdfree example */

```
#include <stdio.h>
#include <dos.h>
#include <process.h>

int main(void)
{
  struct diskfree_t free;
  long avail;

  if (_dos_getdiskfree(0, &free) != 0) {
    printf("Error in _dos_getdiskfree() call\n");
    exit(1);
  }
  avail = (long) free.avail_clusters
        * (long) free.bytes_per_sector
        * (long) free.sectors_per_cluster;
  printf("The current drive has %ld bytes available\n", avail);
  return 0;
}
```

/* _chdrive example */

```
 #include <stdio.h>
#include <direct.h>

int main(void)
{
  if (_chdrive(3) == 0)
    printf("Successfully changed to drive C:\n");
  else
    printf("Cannot change to drive C:\n");
  return 0;
}
```

```
/* _dos_getdrive example */
#include <stdio.h>
#include <dos.h>

int main(void)
{
  unsigned disk;
  _dos_getdrive(&disk);
  printf("The current drive is: %c\n", disk + 'A' - 1);
  return 0;
}
```

```
/* _dos_setdrive example */
#include <stdio.h>
#include <dos.h>

int main(void)
{
  unsigned maxdrives;
  _dos_setdrive(3,&maxdrives);     /* set drive to C: */
  printf("The number of logical drives is: %d\n", maxdrives);
  return 0;
}
```

```
/* _getdrive example */
#include <stdio.h>
#include <direct.h>

int main(void)
{
  int disk;
  disk = _getdrive() + 'A' - 1;
  printf("The current drive is: %c\n", disk);
  return 0;
}
```

/* getdisk example */

```
#include <stdio.h>
#include <dir.h>

int main(void)
{
  int disk, maxdrives = setdisk(2);
  disk = getdisk() + 'A';
  printf("\nThe number of logical drives is:%d\n", maxdrives);
  printf("The current drive is: %c\n", disk);
  return 0;
 }
```

/* setdisk example */

```
#include <stdio.h>
#include <dir.h>

int main(void)
{
   int save, disk, disks;

   /* save original drive */
   save = getdisk();

   /* print number of logic drives */
   disks = setdisk(save);
   printf("%d logical drives on the system\n\n", disks);

   /* print the drive letters available */
   printf("Available drives:\n");
   for (disk = 0;disk < 26;++disk)
   {
      setdisk(disk);
      if (disk == getdisk())
         printf("%c: drive is available\n", disk + 'a');
   }
   setdisk(save);

   return 0;
}
```

```
/* getdta example */
#include <dos.h>
#include <stdio.h>

int main(void)
{
   char far *dta;

   dta = getdta();
   printf("The current disk transfer address is: %Fp\n", dta);
   return 0;
}
```

/* setdta example */

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>

int main(void)
{
   char line[80], far *save_dta;
   char buffer[256] = "SETDTA test!";
   struct fcb blk;
   int result;

   /* get new file name from user */
   printf("Enter a file name to create:");
   gets(line);

   /* parse the new file name to the dta */
   parsfnm(line, &blk, 1);
   printf("%d %s\n", blk.fcb_drive, blk.fcb_name);

   /* request DOS services to create file */
   if (bdosptr(0x16, &blk, 0) == -1)
   {
      perror("Error creating file");
      exit(1);
   }

   /* save old dta and set new dta */
   save_dta = getdta();
   setdta(buffer);

   /* write new records */
   blk.fcb_recsize = 256;
   blk.fcb_random = 0L;
   result = randbwr(&blk, 1);
   printf("result = %d\n", result);

   if (!result)
      printf("Write OK\n");
   else
   {
      perror("Disk error");
      exit(1);
   }

   /* request DOS services to close the file */
   if (bdosptr(0x10, &blk, 0) == -1)
   {
      perror("Error closing file");
      exit(1);
   }

   /* reset the old dta */
   setdta(save_dta);
   return 0;
```

```
}
```

/* getfat example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   struct fatinfo diskinfo;
   int flag = 0;

   printf("Please insert disk in drive A\n");
   getchar();

   getfat(1, &diskinfo);
/* get drive information */

   printf("\nDrive A: is ");
   switch((unsigned char) diskinfo.fi_fatid)
   {
      case 0xFD:
        printf("360K low density\n");
        break;

      case 0xF9:
        printf("1.2 Meg high density\n");
        break;

      default:
        printf("unformatted\n");
        flag = 1;
   }

   if (!flag)
   {
      printf("  sectors per cluster %5d\n", diskinfo.fi_sclus);
      printf("   number of clusters %5d\n", diskinfo.fi_nclus);
      printf("     bytes per sector %5d\n", diskinfo.fi_bysec);
   }

   return 0;
}
```

/* getfatd example */

```
#include <stdio.h>
#include <dos.h>

int main()
{
   struct fatinfo diskinfo;

   /* get default drive information */
   getfatd(&diskinfo);
   printf("\nDefault Drive:\n");
   printf("sectors per cluster: %5d\n",diskinfo.fi_sclus);
   printf("FAT ID byte:         %5X\n",diskinfo.fi_fatid & 0xFF);
   printf("number of clusters   %5d\n",diskinfo.fi_nclus);
   printf("bytes per sector     %5d\n",diskinfo.fi_bysec);
   return 0;
}
```

```
/* getftime example */
#include <stdio.h>
#include <io.h>

int main(void)
{
   FILE *stream;
   struct ftime ft;

   if ((stream = fopen("TEST.$$$",
        "wt")) == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   getftime(fileno(stream), &ft);
   printf("File time: %u:%u:%u\n",
          ft.ft_hour, ft.ft_min,
          ft.ft_tsec * 2);
   printf("File date: %u/%u/%u\n",
          ft.ft_month, ft.ft_day,
          ft.ft_year+1980);
   fclose(stream);
   return 0;
}
```

/* setftime example */

```
#include <stdio.h>
#include <process.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   struct ftime filet;
   FILE *fp;

   if ((fp = fopen("TEST.$$$", "w")) == NULL)
   {
      perror("Error:");
      exit(1);
   }

   fprintf(fp, "testing...\n");

   /* load ftime structure with new time and date */
   filet.ft_tsec = 1;
   filet.ft_min = 1;
   filet.ft_hour = 1;
   filet.ft_day = 1;
   filet.ft_month = 1;
   filet.ft_year = 21;

   /* show current directory for time and date */
   system("dir TEST.$$$");

   /* change the time and date stamp*/
   setftime(fileno(fp), &filet);

   /* close and remove the temporary file */
   fclose(fp);

   system("dir TEST.$$$");

   unlink("TEST.$$$");
   return 0;
}
```

```
/* _dos_getftime example */
#include <stdio.h>
#include <dos.h>

int main()
{
  FILE *stream;
  unsigned date, time;
  if ((stream = fopen("TEST.$$$", "w")) == NULL)
  {
    fprintf(stderr, "Cannot open output file.\n");
    return 1;
  }
  _dos_getftime(fileno(stream), &date, &time);
  printf("File date: 0x%x\n",date);
  printf("File time: 0x%x\n",time);
  fclose(stream);
  return 0;
}
```

/* _dos_setftime example */

```
#include <stdio.h>
#include <dos.h>

int main()
{
  FILE *stream;
  unsigned date, time;
  if ((stream = fopen("TEST.$$$", "w")) == NULL)
  {
    fprintf(stderr, "Cannot open output file.\n");
    return 1;
  }
  _dos_getftime(fileno(stream), &date, &time);
  printf("File year of TEST.$$$: %d\n",((date >> 9) & 0x7f) + 1980);
  date = (date & 0x1ff) | (21 << 9);
  _dos_setftime(fileno(stream), date, time);
  printf("Set file year to 2001.\n");
  fclose(stream);
  return 0;
}
```

/* puts example */

```
#include <stdio.h>

int main(void)
{
   char string[] = "This is an example output string\n";

   puts(string);
   return 0;
}
```

/* gets example */

```
#include <stdio.h>

int main(void)
{
   char string[80];

   printf("Input a string:");
   gets(string);
   printf("The string input was: %s\n", string);
   return 0;
}
```

/* puttext example */

```
#include <conio.h>

int main(void)
{
   char buffer[512];

   /* put some text to the console */
   clrscr();
   gotoxy(20, 12);
   cprintf("This is a test.  Press any key to continue ...");
   getch();

   /* grab screen contents */
   gettext(20, 12, 36, 21,buffer);
   clrscr();

   /* put selected characters back to the screen */
   gotoxy(20, 12);
   puttext(20, 12, 36, 21, buffer);
   getch();

   return 0;
}
```

```
/* gettext example */
#include <conio.h>

char buffer[4096];

int main(void)
{
   int i;

   clrscr();
   for (i = 0; i <= 20; i++)
      cprintf("Line #%d\r\n", i);
   gettext(1, 1, 80, 25, buffer);

   gotoxy(1, 25);
   cprintf("Press any key to clear screen...");
   getch();
   clrscr();
   gotoxy(1, 25);
   cprintf("Press any key to restore screen...");
   getch();
   puttext(1, 1, 80, 25, buffer);
   gotoxy(1, 25);
   cprintf("Press any key to quit...");
   getch();

   return 0;
}
```

```
/* _dos_gettime example */
#include   <dos.h>

int main(void)
{
  struct dostime_t t;
  _dos_gettime(&t);
  printf("The current time is: %2d:%02d:%02d.%02d\n", t.hour, t.minute,
    t.second, t.hsecond);
  return 0;
}
```

```
/* _dos_settime example */
#include <dos.h>
#include <process.h>
#include <stdio.h>

int main(void)
{
  struct dostime_t reset;
  reset.hour    = 17;
  reset.minute  = 0;
  reset.second  = 0;
  reset.hsecond = 0;
  printf("Setting time to 5 PM.\n");
  _dos_settime(&reset);
  system("time");
  return 0;
}
```

/* gettime example */

```
#include   <stdio.h>
#include   <dos.h>

int main(void)
{
   struct  time t;

   gettime(&t);
   printf("The current time is: %2d:%02d:%02d.%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
   return 0;
}
```

/* settime example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   struct  time t;

   gettime(&t);
   printf("The current minute is: %d\n", t.ti_min);
   printf("The current hour is: %d\n", t.ti_hour);
   printf("The current hundredth of a second is: %d\n", t.ti_hund);
   printf("The current second is: %d\n", t.ti_sec);

   /* Add one to the minutes struct element and then call settime  */
   t.ti_min++;
   settime(&t);

   return 0;
}
```

/* _dos_getvect and _dos_setvect example */

```
#include <stdio.h>
#include <dos.h>

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt get_out(__CPPARGS);    /* interrupt prototype */
void interrupt (*oldfunc)(__CPPARGS); /* interrupt function pointer */


int looping = 1;

int main(void)
{
  puts("Press <Shift><PrtSc> to terminate");

  /* save the old interrupt */
  oldfunc  = _dos_getvect(5);

  /* install interrupt handler */
  _dos_setvect(5,get_out);

  /* do nothing */
  while (looping);

  /* restore to original interrupt routine */
  _dos_setvect(5,oldfunc);

  puts("Success");
  return 0;
}

void interrupt get_out(__CPPARGS) {
  looping = 0;             /* change global var to get out of oop */
}
```

/* getvect and setvect example */

```
/* * * * * * * * * * * * * * * *
NOTE: This is an interrupt service routine.
You can NOT compile this program with
Test Stack Overflow turned on and get an
executable file that will operate correctly.
 * * * * * * * * * * * * * * * */
#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    /* The clock tick interrupt */

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

void interrupt ( *oldhandler)(__CPPARGS);

int count=0;

void interrupt handler(__CPPARGS)
{
/* increase the global counter */
   count++;

/* call the old routine */
  oldhandler();
}
  int main(void)
{
/* save the old interrupt vector */
   oldhandler = getvect(INTR);

/* install the new interrupt handler */
   setvect(INTR, handler);

/* loop until the counter exceeds 20 */
  while (count < 20)
      printf("count is %d\n",count);

/* reset the old interrupt handler */
   setvect(INTR, oldhandler);

   return 0;
}
```

/* getw example */

```
#include <stdio.h>
#include <stdlib.h>

#define FNAME "test.$$$"

int main(void)
{
  FILE *fp;
  int word;

  /* place the word in a file */
  fp = fopen(FNAME, "wb");
  if (fp == NULL)
  {
     printf("Error opening file %s\n", FNAME);
      exit(1);
   }

   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);

   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }

   /* extract the word */
   word = getw(fp);

   if (ferror(fp))
       printf("Error reading file\n");
   else
       printf("Successful read: word = %d\n", word);

   /* clean up */
   fclose(fp);
   unlink(FNAME);

   return 0;
}
```

/* putw example */

```
#include <stdio.h>
#include <stdlib.h>

#define FNAME "test.$$$"

int main(void)
{
   FILE *fp;
   int word;

   /* place the word in a file */
   fp = fopen(FNAME, "wb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }

   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);

   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }

   /* extract the word */
   word = getw(fp);
   if (ferror(fp))
       printf("Error reading file\n");
   else
       printf("Successful read: word = %d\n", word);

   /* clean up */
   fclose(fp);
   unlink(FNAME);

   return 0;
}
```

/* gmtime example */

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>

/* Pacific Standard Time & Daylight Savings */
char *tzstr = "TZ=PST8PDT";

int main(void)
{
   time_t t;
   struct tm *gmt, *area;

   putenv(tzstr);
   tzset();

   t = time(NULL);
   area = localtime(&t);
   printf("Local time is: %s", asctime(area));
   gmt = gmtime(&t);
   printf("GMT is:        %s", asctime(gmt));
   return 0;
}
```

/* localtime example */

```
#include <time.h>
#include <stdio.h>
#include <dos.h>

int main(void)
{
   time_t timer;
   struct tm *tblock;

   /* gets time of day */
   timer = time(NULL);

   /* converts date/time to a structure */
   tblock = localtime(&timer);

   printf("Local time is: %s", asctime(tblock));

   return 0;
}
```

```
/* heapcheck and _heapchk example */
#include <stdio.h>
#include <alloc.h>

#define NUM_PTRS  10
#define NUM_BYTES 16

int main(void)
{
   char *array[ NUM_PTRS ];
   int i;

   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );

   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );

   if( heapcheck() == _HEAPCORRUPT )
      printf( "Heap is corrupted.\n" );
   else
      printf( "Heap is OK.\n" );

   return 0;
}
```

/* farheapcheck example */

```
#include <stdio.h>
#include <alloc.h>

#define NUM_PTRS  10
#define NUM_BYTES 16

int main(void)
{
char far *array[ NUM_PTRS ];
int i;

  for( i = 0; i < NUM_PTRS; i++ )
  array[ i ] = (char far *) farmalloc( NUM_BYTES );

  for( i = 0; i < NUM_PTRS; i += 2 )
  farfree( array[ i ] );

  if( farheapcheck() == _HEAPCORRUPT )
    printf( "Heap is corrupted.\n" );
  else
    printf( "Heap is OK.\n" );

  return 0;
}
```

/* heapchecknode example */

```
#include <stdio.h>
#include <alloc.h>

#define NUM_PTRS  10
#define NUM_BYTES 16

int main(void)
{
   char *array[ NUM_PTRS ];
   int i;

   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );

   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );

   for( i = 0; i < NUM_PTRS; i++ )
   {
      printf( "Node %2d ", i );
      switch( heapchecknode( array[ i ] ) )
      {
         case _HEAPEMPTY:
              printf( "No heap.\n" );
              break;
         case _HEAPCORRUPT:
              printf( "Heap corrupt.\n" );
              break;
         case _BADNODE:
              printf( "Bad node.\n" );
              break;
         case _FREEENTRY:
              printf( "Free entry.\n" );
              break;
         case _USEDENTRY:
              printf( "Used entry.\n" );
              break;
         default:
              printf( "Unknown return code.\n" );
               break;
      }
   }

   return 0;
}
```

/* heapfillfree and heapcheckfree example */

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>

#define NUM_PTRS  10
#define NUM_BYTES 16

int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   int res;

   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );

   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );

   if( heapfillfree( 1 ) < 0 )
   {
      printf( "Heap corrupted.\n" );
      return 1;
   }

   for( i = 1; i < NUM_PTRS; i += 2 )
      memset( array[ i ], 0, NUM_BYTES );

   res = heapcheckfree( 1 );
   if( res < 0 )
      switch( res )
      {
         case _HEAPCORRUPT:
              printf( "Heap corrupted.\n" );
              return 1;
         case _BADVALUE:
              printf( "Bad value in free space.\n" );
              return 1;
         default:
              printf( "Unknown error.\n" );
              return 1;
      }

   printf( "Test successful.\n" );
   return 0;
}
```

/* heapwalk example*/

```
#include <stdio.h>
#include <alloc.h>

#define NUM_PTRS  10
#define NUM_BYTES 16

int main( void )
{
   struct heapinfo hi;
  char *array[ NUM_PTRS ];
   int i;

   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );

   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );

   hi.ptr = NULL;
   printf( "   Size   Status\n" );
   printf( "   ----   ------\n" );
   while( heapwalk( &hi ) == _HEAPOK )
   printf( "%7u    %s\n", hi.size, hi.in_use ? "used" : "free" );

   return 0;
}
```

/* _rtl_heapwalk example*/

```
#include <stdio.h>
#include <malloc.h>
#include <alloc.h>

#define NUM_PTRS  10
#define NUM_BYTES 16
#if defined(__FLAT__)
int main( void )
{
  struct heapinfo hi;
  char *array[ NUM_PTRS ];
  int i;

  for( i = 0; i < NUM_PTRS; i++ )
     array[ i ] = (char *) malloc( NUM_BYTES );

  for( i = 0; i < NUM_PTRS; i += 2 )
     free( array[ i ] );

  hi.ptr = NULL;
  printf( "   Size   Status\n" );
  printf( "   ----   ------\n" );
  while( _rtl_heapwalk( &hi ) == _HEAPOK )
  printf( "%7u    %s\n", hi.size, hi.in_use ? "used" : "free" );

  return 0;
}
#endif
```

/* inp example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   int result;
   int port = 0;  /* serial port 0 */

   result = inport(port);
   printf("Word read from port %d = 0x%X\n", port, result);
   return 0;
}
```

/* inpw example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  unsigned result;
  unsigned port = 0;
  result = inpw(port);
  printf("Word read from port %d = 0x%X\n", port, result);
  return 0;
}
```

/* outp example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  unsigned port = 0;
  int value;
  value = outp(port, 'C');
  printf("Value %c sent to port number %d\n", value, port);
  return 0;
}
```

/* outpw example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  unsigned value;
  unsigned port = 0;
  value = outpw(port, 64);
  printf("Value %d sent to port number %d\n", value, port);
  return 0;
}
```

/* inport example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
  int result;
  int port = 0;
  result = inport(port);
  printf("Word read from port %d = 0x%X\n", port, result);
  return 0;
}
```

/* inportb example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   unsigned char result;
   int port = 0;           /* serial port 1 */

   result = inportb(port);
   printf("Byte read from port %d = 0x%X\n", port, result);
   return 0;
}
```

/* outport example */

```
#include <conio.h>
#include <stdio.h>
int main(void)
{
   int port = 0;
   int value = 'C';

   outport(port, value);
   printf("Value %d sent to port number %d\n", value, port);
   return 0;
}
```

/* outportb example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
  int port = 0;
  char value = 'C';

  outportb(port, value);
   printf("Value %c sent to port number %d\n", value, port);
  return 0;
}
```

/* int86 example */

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define VIDEO 0x10

void movetoxy(int x, int y)
{
  union REGS regs;

   regs.h.ah = 2;  /* set cursor position */
  regs.h.dh = y;
   regs.h.dl = x;
   regs.h.bh = 0;  /* video page 0 */
  int86(VIDEO, &regs, &regs);
}

int main(void)
{
  clrscr();
  movetoxy(35, 10);
  printf("Hello\n");
  return 0;
}
```

/* int86x example */

```
#include <dos.h>
#include <process.h>
#include <stdio.h>

int main(void)
{
  char filename[80];
  union REGS inregs, outregs;
  struct SREGS segregs;

  printf("Enter filename: ");
  gets(filename);
  inregs.h.ah = 0x43;
  inregs.h.al = 0x21;
  inregs.x.dx = FP_OFF(filename);
  segregs.ds = FP_SEG(filename);
  int86x(0x21, &inregs, &outregs, &segregs);
   printf("File attribute: %X\n", outregs.x.cx);
   return 0;
}
```

/* intdos example */

```
#include <stdio.h>
#include <dos.h>

/* deletes file name; returns 0 on success, nonzero on failure */
int delete_file(char near *filename)
{
   union REGS regs;
   int ret;
   regs.h.ah = 0x41;
/* delete file */
   regs.x.dx = (unsigned) filename;
   ret = intdos(&regs, &regs);

   /* if carry flag is set, there was an error */
   return(regs.x.cflag ? ret : 0);
}

int main(void)
{
   int err;
   err = delete_file("NOTEXIST.$$$");
   if (!err)
      printf("Able to delete NOTEXIST.$$$\n");
   else
      printf("Not Able to delete NOTEXIST.$$$\n");
   return 0;
}
```

/* intdosx example */

```
#include <stdio.h>
#include <dos.h>

/* deletes file name; returns 0 on success,
nonzero on failure */
int delete_file(char far *filename)
{
   union REGS regs; struct SREGS sregs;
   int ret;
   regs.h.ah = 0x41;                        /* delete file */
   regs.x.dx = FP_OFF(filename);
   sregs.ds = FP_SEG(filename);
   ret = intdosx(&regs, &regs, &sregs);

   /* if carry flag is set, there was an error */
   return(regs.x.cflag ? ret : 0);
}

int main(void)

{
   int err;
   err = delete_file("NOTEXIST.$$$");
   if (!err)
      printf("Able to delete NOTEXIST.$$$\n");
   else
      printf("Not Able to delete NOTEXIST.$$$\n");
   return 0;
}
```

/* itoa example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int number = 12345;
   char string[25];

   itoa(number, string, 10);
   printf("integer = %d string = %s\n", number, string);
   return 0;
}
```

/* ltoa example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char string[25];
   long value = 123456789L;

   ltoa(value,string,10);
   printf("number = %ld  string = %s\n", value, string);

   return 0;
}
```

/* ultoa example */

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   unsigned long lnumber = 3123456789L;
   char string[25];

   ultoa(lnumber,string,10);
   printf("string = %s  unsigned long = %lu\n",string,lnumber);

   return 0;
}
```

/* keep example */

```
/* * * * * * * * * * * * * *

NOTE: This is an interrupt service routine. You can NOT compile this
  program with Test Stack Overflow turned on and get an executable file
  which will operate correctly.

Due to the nature of this function the formula used to compute the number
  of paragraphs may not necessarily work in all cases.  Use with care!

Terminate Stay Resident (TSR) programs are complex and no other support for
  them is provided.

Refer to the MS-DOS technical documentation for more information.

 * * * * * * * * * * * * * */

#include <dos.h>
/* The clock tick interrupt */
#define INTR 0x1C
/* Screen attribute (blue on grey) */
#define ATTR 0x7900

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

/* reduce heaplength and stacklength to make a smaller program in memory */
extern unsigned _heaplen = 1024;
extern unsigned _stklen  = 512;

void interrupt ( *oldhandler)(__CPPARGS);

typedef unsigned int (far *s_arrayptr);

void interrupt handler(__CPPARGS)
{
   s_arrayptr screen[80];
   static int count;

/* For a color screen the video memory is at B800:0000.
   For a monochrome system use B000:000 */
   screen[0] = (s_arrayptr) MK_FP(0xB800,0);

/* increase the counter and keep it within 0 to 9 */
   count++;
   count %= 10;

/* put the number on the screen */
   screen[0][79] = count + '0' + ATTR;

/* call the old interrupt handler */
   oldhandler();
}
```

```
int main(void)
{

/* get the address of the current clock
   tick interrupt */
oldhandler = getvect(INTR);

/* install the new interrupt handler */
setvect(INTR, handler);

/* * *
_psp is the starting address of the program in memory.  The top of the
  stack is the end of the program.

Using _SS and _SP together we can get the end of the stack.  You may want
  to allow a bit of safety space to insure that enough room is being
  allocated ie:

   (_SS + ((_SP + safety space)/16) - _psp)
* * */

keep(0, (_SS + (_SP/16) - _psp));
return 0;
}
```

/* localeconv example */

```
#include <locale.h>
#include <stdio.h>

int main(void)
{
   struct lconv ll;
   struct lconv *conv = &ll;

/* read the locality conversion structure */
   conv = localeconv();

/* display the structure */
  printf("Decimal Point                : %s\n", conv-> decimal_point);
  printf("Thousands Separator          : %s\n", conv-> thousands_sep);
  printf("Grouping                     : %s\n", conv-> grouping);
  printf("International Currency symbol : %s\n", conv-> int_curr_symbol);
  printf("$ thousands separator        : %s\n", conv-> mon_thousands_sep);
  printf("$ grouping                   : %s\n", conv-> mon_grouping);
  printf("Positive sign                : %s\n", conv-> positive_sign);
  printf("Negative sign                : %s\n", conv-> negative_sign);
  printf("International fraction digits : %d\n", conv-> int_frac_digits);
  printf("Fraction digits              : %d\n", conv-> frac_digits);
  printf("Positive $ symbol precedes   : %d\n", conv-> p_cs_precedes);
  printf("Positive sign space separation: %d\n", conv-> p_sep_by_space);
  printf("Negative $ symbol precedes   : %d\n", conv-> n_cs_precedes);
  printf("Negative sign space separation: %d\n", conv-> n_sep_by_space);
  printf("Positive sign position       : %d\n", conv-> p_sign_posn);
  printf("Negative sign position       : %d\n", conv-> n_sign_posn);
  return 0;
}
```

/* setlocale example */

```
#include <locale.h>
#include <stdio.h>

int main(void)
{
   char *old_locale;

   /* The only locale supported in Borland C++ is "C" */
   old_locale = setlocale(LC_ALL,"C");
   printf("Old locale was %s\n",old_locale);

   return 0;
}
```

/* locking example */

```
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
#include <sys\locking.h>

int main(void)
{
  int handle, status;
  long length;

  /* must have DOS SHARE.EXE loaded for file locking to function */
  handle = sopen("c:\\autoexec.bat", O_RDONLY,SH_DENYNO);
  if (handle < 0) {
    printf("sopen failed\n");
    exit(1);
  }
  length = filelength(handle);
  status = locking(handle,LK_LOCK,length/2);
  if (status == 0)
    printf("lock succeeded\n");
  else
    perror("lock failed");
  status = locking(handle,LK_UNLCK,length/2);
  if (status == 0)
    printf("unlock succeeded\n");
  else
    perror("unlock failed");
  close(handle);
  return 0;
}
```

/* lock example */

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>

int main(void)
{
   int handle, status;
   long length;

   /* Must have DOS Share.exe loaded for */
   /* file locking to function properly */

   handle = sopen("c:\\autoexec.bat",
      O_RDONLY,SH_DENYNO,S_IREAD);

   if (handle < 0)
   {
      printf("sopen failed\n");
      exit(1);
   }

   length = filelength(handle);
   status = lock(handle,0L,length/2);

   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");

   status = unlock(handle,0L,length/2);

   if (status == 0)
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");

   close(handle);
   return 0;
}
```

/* unlock example */

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>

int main(void)
{
   int handle, status;
   long length;

   handle = sopen("c:\\autoexec.bat",O_RDONLY,SH_DENYNO,S_IREAD);

   if (handle < 0)
   {
       printf("sopen failed\n");
       exit(1);
   }

   length = filelength(handle);
   status = lock(handle,0L,length/2);

   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");

   status = unlock(handle,0L,length/2);

   if (status == 0)
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");

   close(handle);
   return 0;
}
```

/* log example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double result;
   double x = 8.6872;

   result = log(x);
   printf("The natural log of %lf is %lf\n", x, result);

   return 0;
}
```

/* log10 example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double result;
   double x = 800.6872;

   result = log10(x);
   printf("The common log of %lf is %lf\n", x, result);

   return 0;
}
```

/* _lrotl and _lrotr example */

```
#include <stdlib.h>
#include <stdio.h>

/* function prototypes */

int lrotl_example(void);
int lrotr_example(void);

/* lrotl example */

int lrotl_example(void)
{
   unsigned long result;
   unsigned long value = 100;

   result = _lrotl(value,1);
   printf("The value %lu rotated left one bit is: %lu\n", value, result);

   return 0;
}

/* lrotr example */

int lrotr_example(void)
{
   unsigned long result;
   unsigned long value = 100;

   result = _lrotr(value,1);
   printf("The value %lu rotated right one bit is: %lu\n", value, result);

   return 0;
}

int main(void)
{
   lrotl_example();
   lrotr_example();
   return 0;
}
```

/* _rotl and _rotr example */

```
#include <stdlib.h>
#include <stdio.h>

/* rotl example */

int rotl_example(void)
{
   unsigned value, result;

   value = 32767;
   result = _rotl(value, 1);
   printf("The value %u rotated left one bit is: %u\n", value, result);
   return 0;
}

/* rotr example */

int rotr_example(void)
{
   unsigned value, result;

   value = 32767;
   result = _rotr(value, 1);
   printf("The value %u rotated right one bit is: %u\n", value, result);
   return 0;
}

int main(void)
{
   rotl_example();
   rotr_example();
   return 0;
}
```

/* _makepath example */

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];

  getcwd(s,_MAX_PATH);            /* get current working directory */
 if (s[strlen(s)-1] != '\\')
     strcat(s,"\\");              /* append a trailing \ character */
  _splitpath(s,drive,dir,file,ext); /* split the string to separate
    elems */
  strcpy(file,"DATA");
  strcpy(ext,".TXT");
  _makepath(s,drive,dir,file,ext); /* merge everything into one string */
  puts(s);                        /* display resulting string */
  return 0;
}
```

/* _splitpath example */

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];

  /* get current working directory */
  getcwd(s,_MAX_PATH);
  if (s[strlen(s)-1] != '\\')

  /* append a trailing \ character */
    strcat(s,"\\");

  /* split the string to separate elems */
  _splitpath(s,drive,dir,file,ext);
  strcpy(file,"DATA");
  strcpy(ext,".TXT");

  /* merge everything into one string */
  _makepath(s,drive,dir,file,ext);

  /* display resulting string */
  puts(s);
  return 0;
}
```

/* fnsplit example */

```
#include <stdlib.h>
#include <stdio.h>
#include <dir.h>

int main(void)
{
    char *s;
    char drive[MAXDRIVE];
    char dir[MAXDIR];
    char file[MAXFILE];
    char ext[MAXEXT];
    int flags;

    s=getenv("COMSPEC"); /* get the comspec environment parameter */
    flags=fnsplit(s,drive,dir,file,ext);

    printf("Command processor info:\n");
    if(flags & DRIVE)
       printf("\tdrive: %s\n",drive);
    if(flags & DIRECTORY)
       printf("\tdirectory: %s\n",dir);
    if(flags & FILENAME)
       printf("\tfile: %s\n",file);
    if(flags & EXTENSION)
       printf("\textension: %s\n",ext);

    return 0;
}
```

/* fnmerge example */

```
#include <string.h>
#include <stdio.h>
#include <dir.h>

int main(void)
{
   char s[MAXPATH];
   char drive[MAXDRIVE];
   char dir[MAXDIR];
   char file[MAXFILE];
   char ext[MAXEXT];

   getcwd(s,MAXPATH);              /* get the current working directory */
   strcat(s,"\\");                 /* append on a trailing character */
   fnsplit(s,drive,dir,file,ext);  /* split the string to separate elems */
   strcpy(file,"DATA");
   strcpy(ext,".TXT");
   fnmerge(s,drive,dir,file,ext);   /* merge everything into one string */
   puts(s);                         /* display resulting string */

   return 0;
}
```

/* memmove example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *dest = "abcdefghijklmnopqrstuvwxyz0123456789";
  char *src = "******************************";
  printf("destination prior to memmove: %s\n", dest);
  memmove(dest, src, 26);
  printf("destination after memmove:    %s\n", dest);
  return 0;
}
```

/* memccpy example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *src = "This is the source string";
   char dest[50];
   char *ptr;

   ptr = (char *) memccpy(dest, src, 'c', strlen(src));

   if (ptr)
   {
      *ptr = '\0';
      printf("The character was found: %s\n", dest);
   }
   else
      printf("The character wasn't found\n");
   return 0;
}
```

/* memcpy example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char src[] = "******************************";
   char dest[] = "abcdefghijlkmnopqrstuvwxyz0123456709";
   char *ptr;

   printf("destination before memcpy: %s\n", dest);
   ptr = (char *) memcpy(dest, src, strlen(src));
   if (ptr)
      printf("destination after memcpy: %s\n", dest);
   else
      printf("memcpy failed\n");
   return 0;
}
```

/* memcmp example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *buf1 = "aaa";
   char *buf2 = "bbb";
   char *buf3 = "ccc";

   int stat;

   stat = memcmp(buf2, buf1, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");

   stat = memcmp(buf2, buf3, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");

   return 0;
}
```

/* memicmp example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *buf1 = "ABCDE123";
   char *buf2 = "abcde456";
   int stat;
   stat = memicmp(buf1, buf2, 5);
   printf("The strings to position 5 are ");
   if (stat)
      printf("not ");
   printf("the same\n");
   return 0;
}
```

/* _dos_open example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <dos.h>

int main(void)
{
  int handle;
  unsigned nbytes;
  char msg[] = "Hello world\n";
  if (_dos_open("TEST.$$$", O_RDWR, &handle) != 0) {
    perror("Unable to open TEST.$$$");
    return 1;
  }
  if (_dos_write(handle, msg, strlen(msg),&nbytes) != 0)
    perror("Unable to write to TEST.$$$");
  printf("%u bytes written to TEST.$$$\n",nbytes);
  _dos_close(handle);
  return 0;
}
```

```
/* _rtl_open example */
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char msg[] = "Hello world";

   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

/* sopen example */

```
/*      Load share before running this example.
*/
 #include <io.h>
 #include <fcntl.h>
 #include <sys\stat.h>
 #include <process.h>
 #include <share.h>
 #include <stdio.h>
 #include       <stdlib.h>

 int main(void)
 {
    int handle,
        handle1;

    handle = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);

    if      (handle == -1)
    {
      perror (sys_errlist[errno]);
      exit (1);
    }

    if (!handle)
    {
       printf("sopen failed\n");
       exit(1);
    }

    /*      Attempt sopen for write.
    */
    handle1 = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);

    if      (handle1 == -1)
    {
      perror (sys_errlist[errno]);
      exit (1);
    }

    if (!handle1)
    {
       printf("sopen failed\n");
       exit(1);
    }

    close (handle);
    close (handle1);
    return 0;
}
```

/* open example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char msg[] = "Hello world";

   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   close(handle);
   return 0;
}
```

/* cprintf example */

```
#include <conio.h>

int main(void)
{
   /* clear the screen */
   clrscr();

   /* create a text window */
   window(10, 10, 80, 25);

   /* output some text in the window */
   cprintf("Hello world\r\n");

   /* wait for a key */
   getch();
   return 0;
}
```

/* fprintf example */

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int i = 100;
   char c = 'C';
   float f = 1.234;

   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");

   /* write some data to the file */
   fprintf(stream, "%d %c %f", i, c, f);

   /* close the file */
   fclose(stream);
   return 0;
}
```

/* printf example */

```
#include <stdio.h>
#include <string.h>

#define I 555
#define R 5.5

int main(void)
{
   int i,j,k,l;
   char buf[7];
   char *prefix = buf;
   char tp[20];
   printf("prefix  6d      6o      8x        10.2e        "
          "10.2f\n");
   strcpy(prefix,"%");
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
         for (k = 0; k < 2; k++)
            for (l = 0; l < 2; l++)
            {
               if (i==0)  strcat(prefix,"-");
               if (j==0)  strcat(prefix,"+");
               if (k==0)  strcat(prefix,"#");
               if (l==0)  strcat(prefix,"0");
               printf("%5s |",prefix);
               strcpy(tp,prefix);
               strcat(tp,"6d |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"6o |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"8x |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"10.2e |");
               printf(tp,R);
               strcpy(tp,prefix);
               strcat(tp,"10.2f |");
               printf(tp,R);
               printf("  \n");
               strcpy(prefix,"%");
            }
       }
   return 0;
}
```

/* sprintf example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   char buffer[80];

   sprintf(buffer, "An approximation of Pi is %f\n", M_PI);
   puts(buffer);
   return 0;
}
```

/* vfprintf example */

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *fp;

int vfpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;

   va_start(argptr, fmt);
   cnt = vfprintf(fp, fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";

   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }

   vfpf("%d %f %s", inumber, fnumber, string);
   rewind(fp);
   fscanf(fp,"%d %f %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   fclose(fp);

   return 0;
}
```

/* vprintf example */

```
#include <stdio.h>
#include <stdarg.h>

int vpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;

   va_start(argptr, fmt);
   cnt = vprintf(fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char *string = "abc";

   vpf("%d %f %s\n",inumber,fnumber,string);

   return 0;
}
```

/* vsprintf example */

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>

char buffer[80];

int vspf(char *fmt, ...)
{
   va_list argptr;
   int cnt;

   va_start(argptr, fmt);
   cnt = vsprintf(buffer, fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";

   vspf("%d %f %s", inumber, fnumber, string);
   printf("%s\n", buffer);
   return 0;
}
```

/* _dos_read example */

```
#include <stdio.h>
#include <fcntl.h>
#include <dos.h>

int main(void)
{
  int handle;
  unsigned bytes;
  char buf[10];

  /* Looks for a file in the current directory named TEST.$$$ and
    attempts to read 10 bytes from it. To use this example you
    should create the file TEST.$$$ */
  if (_dos_open("TEST.$$$", O_RDONLY, &handle) != 0) {
    perror("Unable to open TEST.$$$");
    return 1;
  }
  if (_dos_read(handle, buf, 10, &bytes) != 0) {
    perror("Unable to read from TEST.$$$");
    return 1;
  }
  else
    printf("_dos_read: %d bytes read.\n", bytes);
  return 0;
}
```

/* _rtl_read example */

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>

int main(void)
{
   void *buf;
   int handle, bytes;

   buf = malloc(10);

/*
Looks for a file in the current directory named TEST.$$$ and attempts to
  read 10 bytes from it. To use this example you should create the file
  TEST.$$$
 */
   if ((handle =
       open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }

   if ((bytes = _rtl_read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("_rtl_read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

/* read example */

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>

int main(void)
{
   void *buf;
   int handle, bytes;

   buf = malloc(10);

/*
Looks for a file in the current directory named TEST.$$$ and attempts to
  read 10 bytes from it. To use this example you should create the file
  TEST.$$$.
 */
   if ((handle =
      open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }

   if ((bytes = read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("Read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

/* farrealloc example */

```
#include <stdio.h>
#include <alloc.h>

int main(void)
{
   char far *fptr;
   char far *newptr;

   fptr = (char far *) farmalloc(16);
   printf("First address: %Fp\n", fptr);

/*
We use a second pointer, newptr, so that in the case of farrealloc()
  returning NULL, our original pointer is not set to NULL.
 */

   newptr = (char far *) farrealloc(fptr,64);
   printf("New address  : %Fp\n", newptr);
   if (newptr != NULL)
      farfree(newptr);
   return 0;
}
```

/* realloc example */

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>

int main(void)
{
   char *str;

   /* allocate memory for string */
   str = (char *) malloc(10);

   /* copy "Hello" into string */
   strcpy(str, "Hello");

   printf("String is %s\n  Address is %p\n", str, str);
   str = (char *) realloc(str, 20);
   printf("String is %s\n  New address is %p\n", str, str);

   /* free memory */
   free(str);

   return 0;
}
```

/* cscanf example */

```
#include <conio.h>

int main(void)
{
   char string[80];

   /* clear the screen */
   clrscr();

   /* Prompt the user for input */
   cprintf("Enter a string with no spaces:");

   /* read the input */
   cscanf("%s", string);

   /* display what was read */
   cprintf("\r\nThe string entered is: %s", string);
   return 0;
}
```

/* fscanf example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int i;

   printf("Input an integer: ");

   /* read an integer from the
      standard input stream */
   if (fscanf(stdin, "%d", &i))
      printf("The integer read was: %i\n", i);
   else
   {
      fprintf(stderr, "Error reading an integer from stdin.\n");
      exit(1);
   }
   return 0;
}
```

/* scanf example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char label[20];
   char name[20];
   int entries = 0;
   int loop, age;
   double salary;

   struct Entry_struct
   {
      char  name[20];
      int   age;
      float salary;
   } entry[20];

/* Input a label as a string of characters restricting to 20 characters */
   printf("\n\nPlease enter a label for the chart: ");
   scanf("%20s", label);
   fflush(stdin);  /* flush the input stream in case of bad input */

/* Input number of entries as an integer */
  printf("How many entries will there be? (less than 20) ");
  scanf("%d", &entries);
  fflush(stdin);   /* flush the input stream in case of bad input */

/* input a name restricting input to only letters upper or lower case */
   for (loop=0;loop<entries;++loop)
   {
      printf("Entry %d\n", loop);
      printf("  Name   : ");
      scanf("%[A-Za-z]", entry[loop].name);
      fflush(stdin);  /* flush the input stream in case of bad input */

/* input an age as an integer */
      printf("  Age    : ");
      scanf("%d", &entry[loop].age);
      fflush(stdin);  /* flush the input stream in case of bad input */

/* input a salary as a float */
      printf("  Salary : ");
      scanf("%f", &entry[loop].salary);
      fflush(stdin); /* flush the input stream in case of bad input */
   }

/* Input a name, age and salary as a string, integer, and double */
   printf("\nPlease enter your name, age and salary\n");
   scanf("%20s %d %lf", name, &age, &salary);


/* Print out the data that was input */
   printf("\n\nTable %s\n",label);
   printf("Compiled by %s  age %d  $%15.2lf\n", name, age, salary);
```

```
   printf("-----------------------------------------------------\n");
   for (loop=0;loop<entries;++loop)
      printf("%4d | %-20s | %5d | %15.2lf\n",
         loop + 1,
         entry[loop].name,
         entry[loop].age,
         entry[loop].salary);
   printf("-----------------------------------------------------\n");
   return 0;
}
```

/* sscanf example */

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

char *names[4] = {"Peter", "Mike", "Shea", "Jerry"};

#define NUMITEMS 4

int main(void)
{
   int   loop;
   char  temp[4][80];

   char  name[20];
   int   age;
   long  salary;

/* clear the screen */
   clrscr();

/* create name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      sprintf(temp[loop], "%s %d %ld", names[loop], random(10) + 20,
  random(5000) + 27500L);

/* print title bar */
   printf("%4s | %-20s | %5s | %15s\n", "#", "Name", "Age", "Salary");
   printf("   ------------------------------------------------\n");

/* input a name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      {
      sscanf(temp[loop],"%s %d %ld", &name, &age, &salary);
      printf("%4d | %-20s | %5d | %15ld\n", loop + 1, name, age, salary);
      }

   return 0;
}
```

/* vfscanf example */

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *fp;

int vfsf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;

   va_start(argptr, fmt);
   cnt = vfscanf(fp, fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
         char string[4] = "abc";

   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }
   fprintf(fp,"%d %f %s\n",inumber,fnumber,string);
   rewind(fp);

   vfsf("%d %f %s",&inumber,&fnumber,string);
   printf("%d %f %s\n",inumber,fnumber,string);
   fclose(fp);

   return 0;
}
```

/* vscanf example */

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>

int vscnf(char *fmt, ...)
{
   va_list argptr;
   int cnt;

   printf("Enter an integer, a float, and a string (e.g. i,f,s,)\n");
   va_start(argptr, fmt);
   cnt = vscanf(fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber;
   float fnumber;
   char string[80];

   vscnf("%d, %f, %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);

   return 0;
}
```

/* vsscanf example */

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>

char buffer[80] = "30 90.0 abc";

int vssf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;

   fflush(stdin);

   va_start(argptr, fmt);
   cnt = vsscanf(buffer, fmt, argptr);
   va_end(argptr);

   return(cnt);
}

int main(void)
{
   int inumber;
   float fnumber;
   char string[80];

   vssf("%d %f %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   return 0;
}
```

/* setbuf example */

```
#include <stdio.h>

/* BUFSIZ is defined in stdio.h */
char outbuf[BUFSIZ];

int main(void)
{
   /* attach a buffer to the standard output stream */
   setbuf(stdout, outbuf);

   /* put some characters into the buffer */
   puts("This is a test of buffered output.\n\n");
   puts("This output will go into outbuf\n");
   puts("and won't appear until the buffer\n");
   puts("fills up or we flush the stream.\n");

   /* flush the output buffer */
   fflush(stdout);

   return 0;
}
```

/* setvbuf example */

```
#include <stdio.h>

int main(void)
{
   FILE *input, *output;
   char bufr[512];

   input = fopen("file.in", "r+b");
   output = fopen("file.out", "w");

   /* set up input stream for minimal disk access,
      using our own character buffer */
  if (setvbuf(input, bufr, _IOFBF, 512) != 0)
      printf("failed to set up buffer for input file\n");
   else
      printf("buffer set up for input file\n");

   /* set up output stream for line buffering using space that
      will be obtained through an indirect call to malloc */
   if (setvbuf(output, NULL, _IOLBF, 132) != 0)
      printf("failed to set up buffer for output file\n");
   else
      printf("buffer set up for output file\n");

   /* perform file I/O here */

   /* close files */
   fclose(input);
   fclose(output);
   return 0;
}
```

/* spawnl example */

```
#include <process.h>
#include <stdio.h>
#include <conio.h>

void spawnl_example(void)
{
   int result;

   clrscr();
   result = spawnl(P_WAIT, "bcc.exe", "bcc.exe", NULL);
   if (result == -1)
   {
      perror("Error from spawnl");
      exit(1);
   }
}


int main(void)
{
  spawnl_example();
  return 0;
}
```

/* spawnle example */

```
#include <process.h>
#include <stdio.h>
#include <conio.h>

void spawnle_example(void)
 {
   int result;

   clrscr();
   result = spawnle(P_WAIT, "bcc.exe", "bcc.exe", NULL, NULL);
   if (result == -1)
   {
      perror("Error from spawnle");
      exit(1);
   }
}

int main(void)
{
  spawnle_example();
  return 0;
}
```

/* spawnlp example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

void main(int argc, char *argv[])
{
   int i;

   printf("Command line arguments:\n");
   for (i=0; i<argc; ++i)
      printf("[%2d] : %s\n", i, argv[i]);
   printf("About to exec child with arg1 arg2 ...\n");
   spawnlp(P_WAIT, "C:\\BC5\\BIN\\BCC.EXE", "C:\\BC5\\BIN\\BCC.EXE",
  argv[1], argv[2], NULL);

   perror("exec error");
   exit(1);
}
```

/* spawnlpe example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main( int argc, char *argv[], char **envp )
{
   int i;

   printf("Command line arguments:\n");

   for (i=0; i < argc; ++i)
      printf("[%2d] %s\n", i, argv[i]);

   printf("About to exec child with arg1 arg2 ...\n");
   spawnlpe(P_WAIT, "C:\\BC5\\BIN\\BCC.EXE", "C:\\BC5\\BIN\\BCC.EXE",
  argv[1], argv[2], NULL, envp);

   perror("exec error");
   exit(1);

   return 0;
}
```

/* spawnv example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

void main(int argc, char *argv[])
{
   int i;

   printf("Command line arguments:\n");
   for (i=0; i<argc; ++i)
      printf("[%2d] : %s\n", i, argv[i]);
   printf("About to exec child with arg1 arg2 ...\n");
  spawnv(P_WAIT, "C:\\BC5\\BIN\\BCC.EXE", argv);

   perror("exec error");
   exit(1);
 }
```

/* spawnve example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

void main(int argc, char *argv[], char **envp)
{
   int i;

   printf("Command line arguments:\n");
   for (i=0; i<argc; ++i)
      printf("[%2d] : %s\n", i, argv[i]);
   printf("About to exec child with arg1 arg2 ...\n");
  spawnve(P_WAIT, "C:\\BC5\\BIN\\TDMEM.EXE", argv, envp);

   perror("exec error");
   exit(1);
}
```

/* spawnvp example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

void main(int argc, char *argv[])
{
   int i;

   printf("Command line arguments:\n");
   for (i=0; i<argc; ++i)
      printf("[%2d] : %s\n", i, argv[i]);
   printf("About to exec child with arg1 arg2 ...\n");
   spawnvp(P_WAIT, "C:\\BC5\\BIN\\BCC.EXE", argv);

   perror("exec error");
   exit(1);
}
```

/* spawnvpe example */

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main( int argc, char *argv[], char **envp )
{
  int i;

  printf("Command line arguments:\n");

  for (i=0; i < argc; ++i)
     printf("[%2d] %s\n", i, argv[i]);

  printf("About to exec child with arg1 arg2 ...\n");
  spawnvpe(P_WAIT, "C:\\BC5\\BIN\\BCC.EXE", argv, envp);

  perror("exec error");
  exit(1);

   return 0;
}
```

/* strcmp example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
  int ptr;

  ptr = strcmp(buf2, buf1);
  if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");

   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");

   return 0;
}
```

/* strcmpi example */

```
/* strncmpi example */

#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;

   ptr = strcmpi(buf2, buf1);

   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");

   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");

   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");

   return 0;
}
```

/* stricmp example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;

   ptr = stricmp(buf2, buf1);

   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");

   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");

   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");

   return 0;
}
```

/* _strnextc example */

```
#include <tchar.h>
#include <stdio.h>

int main()
{
   unsigned int retval = 0;
   const unsigned char *string = "ABC";

   retval = _strnextc(string);
   printf("The starting character:%c", retval);

   retval = _strnextc(++string);
   printf("\nThe next character:%c", retval);

   return 0;
}

/***
The starting character:A
The next character:B
***/
```

/* strspn example */

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main(void)
{
   char *string1 = "1234567890";
   char *string2 = "123DC8";
   int length;

   length = strspn(string1, string2);
   printf("Character where strings differ is at position %d\n", length);
   return 0;
}
```

/* strcspn example */

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main(void)
{
   char *string1 = "1234567890";
   char *string2 = "747DC8";
   int length;

   length = strcspn(string1, string2);
   printf("Character where strings intersect is at position %d\n",
          length);

   return 0;
}
```

```
/* _strerror example */
#include <stdio.h>
#include <errno.h>

int main(void)
{
   char *buffer;
   buffer = strerror(errno);
   printf("Error: %s\n", buffer);
   return 0;
}
```

/* strerror example */

```
#include <stdio.h>
#include <errno.h>

int main(void)
{
  char *buffer;
  buffer = strerror(errno);
  printf("Error: %s\n", buffer);
  return 0;
}
```

/* strlwr example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "Borland International";

   printf("string prior to strlwr: %s\n", string);
   strlwr(string);
   printf("string after strlwr:    %s\n", string);
   return 0;
}
```

/* strupr example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz", *ptr;

   /* converts string to upper case characters */
   ptr = strupr(string);
   printf("%s\n", ptr);
   return 0;
}
```

/* strncmp example */

```
#include <string.h>
#include <stdio.h>

int  main(void)

{
   char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
   int ptr;

   ptr = strncmp(buf2,buf1,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");

   ptr = strncmp(buf2,buf3,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");

   return(0);
}
```

/* strncmpi Example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;

   ptr = strncmpi(buf2,buf1,3);

   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");

   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");

   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");

   return 0;
}
```

/* strnicmp Example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;

   ptr = strnicmp(buf2, buf1, 3);

   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");

   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");

   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");

   return 0;
}
```

/* strtod example */

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char input[80], *endptr;
   double value;

   printf("Enter a floating point number:");
   gets(input);
   value = strtod(input, &endptr);
   printf("The string is %s the number is %lf\n", input, value);
   return 0;
}
```

/* strtol example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string = "87654321", *endptr;
   long lnumber;

   /* strtol converts string to long integer  */
   lnumber = strtol(string, &endptr, 10);
   printf("string = %s  long = %ld\n", string, lnumber);

   return 0;
}
```

/* strtoul example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string = "87654321", *endptr;
   unsigned long lnumber;

   lnumber = strtoul(string, &endptr, 10);
   printf("string = %s  long = %lu\n",
   string, lnumber);

   return 0;
}
```

```
/* textattr example */
#include <conio.h>

int main(void)
{
   int i;

   clrscr();
   for (i=0; i<9; i++)
   {
       textattr(i + ((i+1) << 4));
       cprintf("This is a test\r\n");
   }

   return 0;
}
```

/* textbackground and textcolor example */

```
#include <conio.h>

int main(void)
{
   int i, j;

   clrscr();
   for (i=0; i<9; i++)
   {
       for (j=0; j<80; j++)
          cprintf("C");
       cprintf("\r\n");
       textcolor(i+1);
       textbackground(i);
   }

   return 0;
}
```

/* time example */

```
#include <time.h>
#include <stdio.h>
#include <dos.h>

int main(void)
{
   time_t t;

   t = time(NULL);
   printf("The number of seconds since January 1, 1970 is %ld",t);
   return 0;
}
```

/* stime example */

```
#include <stdio.h>
#include <time.h>

int main(void)
{
  time_t t;

  t = time(NULL);

  printf("Current date is %s", ctime(&t));

  t -= 24L*60L*60L;  /* Back up to same time previous day */

  stime(&t);
  printf("\nNew date is %s", ctime(&t));

   return 0;
}
```

/* tolower example */

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING";

   length = strlen(string);
   for (i=0; i<length; i++)
   {
       string[i] = tolower(string[i]);
   }
   printf("%s\n",string);

   return 0;
}
```

/* _tolower example */

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING.";

   length = strlen(string);
   for (i = 0; i < length; i++) {
       if ((string[i] >= 'A') && (string[i] <= 'Z')){
          string[i] = _tolower(string[i]);
       }
   }

   printf("%s\n",string);
   return 0;
}
```

/* toupper example */

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int length, i;
   char *string = "this is a string";

   length = strlen(string);
   for (i=0; i<length; i++)
   {
      string[i] = toupper(string[i]);
   }

   printf("%s\n",string);

   return 0;
}
```

/* _toupper example */

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int length, i;
   char *string = "this is a string.";

   length = strlen(string);
   for (i = 0; i < length; i++) {
      if ((string[i] >= 'a') && (string[i] <= 'z')){
         string[i] = _toupper(string[i]);
         }
   }
   printf("%s\n",string);
   return 0;
}
```

/* _dos_write example */

```
#include <dos.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
  unsigned count;
  int handle;
  char buf[11] = "0123456789";

  /* create a file containing 10 bytes */
  if (_dos_creat("DUMMY.FIL", _A_NORMAL, &handle) != 0)
  {
    perror("Unable to create DUMMY.FIL");
    return 1;
  }
  if (_dos_write(handle, buf, strlen(buf), &count) != 0)
  {
    perror("Unable to write to DUMMY.FIL");
    return 1;
  }
  /* close the file */
  _dos_close(handle);
  return 0;
}
```

/* _rtl_write example */

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>

int main(void)
{
   void *buf;
   int handle, bytes;

   buf = malloc(200);

/*
Create a file name TEST.$$$ in the current directory and writes 200 bytes
  to it. If TEST.$$$ already exists, it's overwritten.
 */

   if ((handle = open("TEST.$$$", O_CREAT | O_WRONLY | O_BINARY,
                      S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }

   if ((bytes = _rtl_write(handle, buf, 200)) == -1) {
      printf("Write Failed.\n");
      exit(1);
   }
   printf("_rtl_write: %d bytes written.\n",bytes);

   return 0;
}
```

/* write example */

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <string.h>

int main(void)
{
   int handle;
   char string[40];
   int length, res;

/*
Create a file named "TEST.$$$" in the current directory and write a string
  to it.  If "TEST.$$$" already exists, it will be overwritten.
 */

   if ((handle = open("TEST.$$$", O_WRONLY | O_CREAT | O_TRUNC,
                        S_IREAD | S_IWRITE)) == -1)
   {
      printf("Error opening file.\n");
      exit(1);
   }

   strcpy(string, "Hello, world!\n");
   length = strlen(string);

   if ((res = write(handle, string, length)) != length)
   {
      printf("Error writing to the file.\n");
      exit(1);
   }
   printf("Wrote %d bytes to the file.\n", res);

   close(handle);
   return 0;
}
```

/* getcurdir example */

```
#include <dir.h>
#include <stdio.h>
#include <string.h>

char *current_directory(char *path)
{
  strcpy(path, "X:\\");      /* fill string with form of response: X:\ */
  path[0] = 'A' + getdisk();    /* replace X with current drive letter */
  getcurdir(0, path+3);  /* fill rest of string with current directory */
  return(path);
}

int main(void)
{
  char curdir[MAXPATH];

  current_directory(curdir);
  printf("The current directory is %s\n", curdir);

  return 0;
}
```

/* getenv example */

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
   char *path, *ptr;
   int i = 0;

   /* get the current path environment */
   ptr = getenv("PATH");

   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");

   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);

   return 0;
}
```

/* putenv example */

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
   char *path, *ptr;
   int i = 0;

   /* get the current path environment */
   ptr = getenv("PATH");

   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");

   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);

   return 0;
}
```

/* getpass example */

```
#include <conio.h>

int main(void)
{
   char *password;

   password = getpass("Input a password:");
   cprintf("The password is: %s\r\n", password);
   return 0;
}
```

/* getpid example */

```
#include <stdio.h>
#include <process.h>

int main()
{
   printf("This program's process identification number (PID) "
          "number is %X\n", getpid());
   printf("Note: under DOS it is the PSP segment\n");
   return 0;
}
```

/* gettextinfo example */

```
#include <conio.h>

int main(void)
 {
   struct text_info ti;
   gettextinfo(&ti);
   cprintf("window left      %2d\r\n",ti.winleft);
   cprintf("window top       %2d\r\n",ti.wintop);
   cprintf("window right     %2d\r\n",ti.winright);
   cprintf("window bottom    %2d\r\n",ti.winbottom);
   cprintf("attribute        %2d\r\n",ti.attribute);
   cprintf("normal attribute %2d\r\n",ti.normattr);
   cprintf("current mode     %2d\r\n",ti.currmode);
   cprintf("screen height    %2d\r\n",ti.screenheight);
   cprintf("screen width     %2d\r\n",ti.screenwidth);
   cprintf("current x        %2d\r\n",ti.curx);
   cprintf("current y        %2d\r\n",ti.cury);
   return 0;
}
```

/* getverify example */

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

int main(void)
{
   int verify_flag;

   printf("Enter 0 to set verify flag off\n");
   printf("Enter 1 to set verify flag on\n");

   verify_flag = getch() - 0;

   setverify(verify_flag);

   if (getverify())
      printf("DOS verify flag is on\n");
   else
      printf("DOS verify flag is off\n");

   return 0;
}
```

/* setverify example */

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

int main(void)
{
    int verify_flag;

    printf("Enter 0 to set verify flag off\n");
    printf("Enter 1 to set verify flag on\n");

    verify_flag = getch() - 0;

    setverify(verify_flag);

    if (getverify())
       printf("DOS verify flag is on\n");
    else
       printf("DOS verify flag is off\n");

    return 0;
}
```

/* gotoxy example */

```
#include <conio.h>

int main(void)
{
    clrscr();
    gotoxy(35, 12);
    cprintf("Hello world");
    getch();
    return 0;
}
```

/* harderr example */

```
/*
This program will trap disk errors and
prompt the user for action. Try running it
with no disk in drive A: to invoke its
functions.
*/
#include <stdio.h>
#include <conio.h>
#include <dos.h>


#define IGNORE  0
#define RETRY   1
#define ABORT   2

int buf[500];

/*
define the error messages for trapping disk problems
*/
 static char *err_msg[] = {
     "write protect",
    "unknown unit",
     "drive not ready",
     "unknown command",
     "data error (CRC)",
     "bad request",
    "seek error",
     "unknown media type",
     "sector not found",
     "printer out of paper",
     "write fault",
     "read fault",
     "general failure",
    "reserved",
     "reserved",
     "invalid disk change"
};

error_win(char *msg)
{
    int retval;

    cputs(msg);

/*
prompt for user to press a key to abort, retry, ignore
*/
    while(1)
    {
  retval= getch();
  if (retval == 'a' || retval == 'A')
  {
      retval = ABORT;
      break;
```

```
  }
  if (retval == 'r' || retval == 'R')
  {
     retval = RETRY;
      break;
  }
  if (retval == 'i' || retval == 'I')
  {
      retval = IGNORE;
      break;
  }
    }

    return(retval);
}

/*
pragma warn -par reduces warnings which occur
due to the non use of the parameters
not_used1 and not_used2 to the handler.
*/
#pragma warn -par
void handler(unsigned int ax, unsigned int not_used1, unsigned int
  *not_used2)
{
  static char msg[80];
  unsigned di;
  int drive;
  int errorno;

  di= _DI;
/*
if this is not a disk error then it was
another device having trouble
*/

  if (ax < 0)
  {
     /* report the error */
     error_win("Device error");
     /* and return to the program directly requesting abort */
     _hardretn(ABORT);
  }
/* otherwise it was a disk error */
  drive = ax & 0x00FF;
  errorno = di & 0x00FF;
/* report which error it was */
  sprintf(msg, "Error: %s on drive %c\r\nA)bort, R)etry, I)gnore: ",
    err_msg[errorno], 'A' + drive);
/*
return to the program via dos interrupt 0x23 with abort, retry,
or ignore as input by the user.
*/
  _hardresume(error_win(msg));
  // return ABORT;
}
#pragma warn +par
```

```
int main(void)
{
/*
install our handler on the hardware problem interrupt
*/
  _harderr(handler);
   clrscr();
   printf("Make sure there is no disk in drive A:\n");
   printf("Press any key ....\n");
   getch();
   printf("Trying to access drive A:\n");
   printf("fopen returned %p\n",fopen("A:temp.dat", "w"));
   return 0;
}
```

/* hardresume example */

```
/*
This program will trap disk errors and
prompt the user for action. Try running it
with no disk in drive A: to invoke its
functions.
*/
#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define IGNORE  0
#define RETRY   1
#define ABORT   2

int buf[500];

/*
define the error messages for trapping disk problems
*/
static char *err_msg[] = {
    "write protect",
   "unknown unit",
   "drive not ready",
    "unknown command",
    "data error (CRC)",
    "bad request",
    "seek error",
    "unknown media type",
    "sector not found",
    "printer out of paper",
    "write fault",
    "read fault",
    "general failure",
   "reserved",
   "reserved",
    "invalid disk change"
};

error_win(char *msg)
{
    int retval;

    cputs(msg);

/*
prompt for user to press a key to abort, retry, ignore
*/
    while(1)
    {
  retval= getch();
  if (retval == 'a' || retval == 'A')
  {
      retval = ABORT;
      break;
  }
```

```
  if (retval == 'r' || retval == 'R')
  {
     retval = RETRY;
     break;
  }
  if (retval == 'i' || retval == 'I')
  {
      retval = IGNORE;
      break;
  }
    }

    return(retval);
}

/*
pragma warn -par reduces warnings which occur
due to the non use of the parameters
not_used1 and not_used2 to the handler.
*/

#pragma warn -par
void handler(unsigned int ax, unsigned int not_used1, unsigned int
  *not_used2)
{
    static char msg[80];
   unsigned di;
   int drive;
   int errorno;

    di= _DI;
 /*
 if this is not a disk error then it was
 another device having trouble
 */

    if (ax < 0)
   {
      /* report the error */
      error_win("Device error");
      /* and return to the program directly requesting abort */
      _hardretn(ABORT);
    }
 /* otherwise it was a disk error */
    drive = ax & 0x00FF;
    errorno = di & 0x00FF;
 /* report which error it was */
   sprintf(msg, "Error: %s on drive %c\r\nA)bort, R)etry, I)gnore: ",
      err_msg[errorno], 'A' + drive);
 /*
 return to the program via dos interrupt 0x23 with abort, retry,
 or ignore as input by the user.
 */
   _hardresume(error_win(msg));
  // return ABORT;
 }
 #pragma warn +par
```

```
 int main(void)
 {
 /*
 install our handler on the hardware problem interrupt
 */
   _harderr(handler);
   clrscr();
   printf("Make sure there is no disk in drive A:\n");
   printf("Press any key ....\n");
    getch();
    printf("Trying to access drive A:\n");
    printf("fopen returned %p\n",fopen("A:temp.dat", "w"));
   return 0;
}
```

```
/* highvideo example */
#include <conio.h>

int main(void)
{
  clrscr();

  lowvideo();
  cprintf("Low Intensity text\r\n");
  highvideo();
  gotoxy(1,2);
  cprintf("High Intensity Text\r\n");

  return 0;
}
```

/* lowvideo example */

```
#include <conio.h>

int main(void)
{
  clrscr();

  highvideo();
   cprintf("High Intensity Text\r\n");
   lowvideo();
  gotoxy(1,2);
   cprintf("Low Intensity Text\r\n");

  return 0;
}
```

/* normvideo example */

```
#include <conio.h>

int main(void)
{
    normvideo();
    cprintf("NORMAL Intensity Text\r\n");
   return 0;
}
```

/* hypot example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 3.0;
    double y = 4.0;

    result = hypot(x, y);
    printf("The hypotenuse is: %lf\n", result);

    return 0;
}
```

/* imag example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
   double x = 3.1, y = 4.2;
    complex z = complex(x,y);
    cout << "z = " << z << "\n";
   cout << "  has real part = " << real(z) << "\n";
    cout << "  and imaginary real part = " << imag(z) << "\n";
    cout << "z has complex conjugate = " << conj(z) << "\n";
   return 0;
}
```

```
/* insline example */
#include <conio.h>

int main(void)
{
    clrscr();
    cprintf("INSLINE inserts an empty line in the text window\r\n");
   cprintf("at the cursor position using the current text\r\n");
    cprintf("background color.  All lines below the empty one\r\n");
    cprintf("move down one line and the bottom line scrolls\r\n");
   cprintf("off the bottom of the window.\r\n");
    cprintf("\r\nPress any key to continue:");
   gotoxy(1, 3);
   getch();
    insline();
    getch();
   return 0;
}
```

/* intr example */

```
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <dos.h>

#define CF 1  /* Carry flag */

int main(void)
{
    char directory[80];
   struct REGPACK reg;

    printf("Enter directory to change to: ");
    gets(directory);
    reg.r_ax = 0x3B << 8;         /* shift 3Bh into  AH */
    reg.r_dx = FP_OFF(directory);
    reg.r_ds = FP_SEG(directory);
    intr(0x21, &reg);
    if (reg.r_flags & CF)
       printf("Directory change failed\n");
    getcwd(directory, 80);
    printf("The current directory is: %s\n", directory);
    return 0;
}
```

/* ioctl example */

```
#include <stdio.h>
#include <dir.h>
#include <io.h>

int main(void)
{
    int stat;

    /* use func 8 to determine if the default drive is removable */
    stat = ioctl(0, 8, 0, 0);
    if (!stat)
       printf("Drive %c is removable.\n", getdisk() + 'A');
    else
       printf("Drive %c is not removable.\n", getdisk() + 'A');
    return 0;
}
```

/* isatty example */

```
#include <stdio.h>
#include <io.h>

int main(void)
{
    int handle;

    handle = fileno(stdprn);
    if (isatty(handle))
       printf("Handle %d is a device type\n", handle);
    else
       printf("Handle %d isn't a device type\n", handle);
    return 0;
}
```

```
/* kbhit example */
#include <conio.h>

int main(void)
{
   cprintf("Press any key to continue:");
    while (!kbhit()) /* do nothing */ ;
    cprintf("\r\nA key was pressed...\r\n");
    return 0;
}
```

/* ldexp example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double value;
    double x = 2;

    /* ldexp raises 2 by a power of 3
       then multiplies the result by 2   */
    value = ldexp(x,3);
    printf("The ldexp value is: %lf\n", value);

    return 0;
}
```

/* setjmp example */

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

void subroutine(jmp_buf);

int main(void)
{

    int value;
    jmp_buf jumper;

    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);

    return 0;
}

void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

/* longjmp example */

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

void subroutine(jmp_buf);

int main(void)
{

    int value;
    jmp_buf jumper;

    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);

    return 0;
}

void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

/* lseek example */

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
    int handle;
    char msg[] = "This is a test";
    char ch;

    /* create a file */
    handle = open("TEST.$$$", O_CREAT | O_RDWR, S_IREAD | S_IWRITE);

    /* write some data to the file */
    write(handle, msg, strlen(msg));

    /* seek to the begining of the file */
    lseek(handle, 0L, SEEK_SET);

    /* reads chars from the file until we hit EOF */
    do
    {
       read(handle, &ch, 1);
       printf("%c", ch);
    }  while (!eof(handle));

    close(handle);
    return 0;
}
```

/* malloc example */

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>

int main(void)
{
    char *str;

    /* allocate memory for string */
    if ((str = (char *) malloc(10)) == NULL)
    {
       printf("Not enough memory to allocate buffer\n");
       exit(1);  /* terminate program if out of memory */
    }

    /* copy "Hello" into string */
    strcpy(str, "Hello");

    /* display string */
    printf("String is %s\n", str);

    /* free memory */
    free(str);

    return 0;
}
```

/* _matherr example */

```
#include <math.h>
#include <string.h>
#include <stdio.h>

int matherr (struct exception *a)
{
  if (a->type == DOMAIN)
    if (!strcmp(a->name,"sqrt")) {
      a->retval = sqrt (-(a->arg1));
    return 1;
    }
  return 0;
}

int main(void)
{
  double x = -2.0, y;
  y = sqrt(x);
  printf("Matherr corrected value: %lf\n",y);
  return 0;
}
```

/* max and min example */

```
#include <stdlib.h>
#include <stdio.h>

#ifdef __cplusplus

   int max (int value1, int value2);

   int max(int value1, int value2)
   {
      return ( (value1 > value2) ? value1 : value2);
   }

#endif

int main(void)
{
    int x = 5;
    int y = 6;
    int z;
    z = max(x, y);
    printf("The larger number is %d\n", z);
    return 0;
}
```

/* memchr example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
    char str[17];
    char *ptr;

    strcpy(str, "This is a string");
    ptr = (char *) memchr(str, 'r', strlen(str));
    if (ptr)
       printf("The character 'r' is at position: %d\n", ptr - str);
    else
       printf("The character was not found\n");
    return 0;
}
```

/* memset example */

```
#include <string.h>
#include <stdio.h>
#include <mem.h>

int main(void)
{
    char buffer[] = "Hello world\n";

    printf("Buffer before memset: %s\n", buffer);
    memset(buffer, '*', strlen(buffer) - 1);
    printf("Buffer after memset:  %s\n", buffer);
    return 0;
}
```

/* mkdir example */

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <dir.h>

#define DIRNAME "testdir.$$$"

int main(void)
{
    int stat;

    stat = mkdir(DIRNAME);
    if (!stat)
           printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }

    getch();
    system("dir/p");
    getch();

    stat = rmdir(DIRNAME);
    if (!stat)
           printf("\nDirectory deleted\n");
    else
    {
           perror("\nUnable to delete directory\n");
       exit(1);
    }

    return 0;
}
```

/* rmdir example */

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <dir.h>

#define DIRNAME "testdir.$$$"

int main(void)
{
    int stat;

    stat = mkdir(DIRNAME);
    if (!stat)
           printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }

    getch();
    system("dir/p");
    getch();

    stat = rmdir(DIRNAME);
    if (!stat)
           printf("\nDirectory deleted\n");
    else
    {
           perror("\nUnable to delete directory\n");
       exit(1);
    }

    return 0;
}
```

/* mktemp example */

```
#include <dir.h>
#include <stdio.h>

int main(void)
{
    /* fname defines the template for the
      temporary file.  */

    char *fname = "TXXXXXX", *ptr;

    ptr = mktemp(fname);
    printf("%s\n",ptr);
    return 0;
}
```

/* mktime example */

```
#include <stdio.h>
#include <time.h>

 char *wday[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday", "Unknown"};

int main(void)
{
    struct tm time_check;
    int year, month, day;

/*  Input a year, month and day to find the weekday for */
    printf("Year:  ");
    scanf("%d", &year);
    printf("Month: ");
    scanf("%d", &month);
    printf("Day:   ");
    scanf("%d", &day);

/*  load the time_check structure with the data */
    time_check.tm_year = year - 1900;
    time_check.tm_mon  = month - 1;
    time_check.tm_mday = day;
    time_check.tm_hour = 0;
    time_check.tm_min  = 0;
    time_check.tm_sec  = 1;
    time_check.tm_isdst = -1;

/*  call mktime to fill in the weekday field of the structure */
    if (mktime(&time_check) == -1)
       time_check.tm_wday = 7;

/*  print out the day of the week */
    printf("That day is a %s\n", wday[time_check.tm_wday]);
    return 0;
}
```

/* modf example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double fraction, integer;
    double number = 100000.567;

    fraction = modf(number, &integer);
    printf("The whole and fractional parts of %lf are %lf and %lf\n",
           number, integer, fraction);
    return 0;
}
```

/* movedata example */

```
#include <mem.h>

#define MONO_BASE 0xB000

char buf[80*25*2];

/* saves the contents of the monochrome screen in buffer */
void save_mono_screen(char near *buffer)
{
   movedata(MONO_BASE, 0, _DS, (unsigned)buffer, 80*25*2);
}

int main(void)
{
   save_mono_screen(buf);
   return 0;
}
```

/* movmem example */

```
#include <mem.h>
#include <alloc.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *source = "Borland International";
    char *destination;
    int length;

    length = strlen(source);
    destination = (char *) malloc(length + 1);
    movmem(source, destination, length);
    printf("%s\n", destination);

    return 0;
}
```

/* movetext example */

```
#include <conio.h>
#include <string.h>

int main(void)
{
    char *str = "This is a test string";

    clrscr();
    cputs(str);
    getch();

    movetext(1, 1, strlen(str), 2, 10, 10);
    getch();

    return 0;
}
```

/* norm example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
    double x = 3.1, y = 4.2;
    complex z = complex(x,y);
    cout << "z = " << z << "\n";
    cout << "  has real part = " << real(z) << "\n";
    cout << "  and imaginary real part = " << imag(z) << "\n";
    cout << "z has complex conjugate = " << conj(z) << "\n";

    double mag = sqrt(norm(z));
    double ang = arg(z);
    cout << "The polar form of z is:\n";
    cout << "   magnitude = " << mag << "\n";
    cout << "   angle (in radians) = " << ang << "\n";
    cout << "Reconstructing z from its polar form gives:\n";
    cout << "   z = " << polar(mag,ang) << "\n";
    return 0;
}
```

/* closedir and readdir example */

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

/* opendir example */

```
 /* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

/* parsfnm example */

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>

int main(void)
{
    char line[80];
    struct fcb blk;

    /* get file name */
    printf("Enter drive and file name (no path; e.g., a:file.dat)\n");
    gets(line);

    /* put file name in fcb */
    if (parsfnm(line, &blk, 1) == NULL)
       printf("Error in parsfm call\n");
    else
       printf("Drive #%d  Name: %11s\n", blk.fcb_drive, blk.fcb_name);

    return 0;
}
```

```
/* peek example */
#include <stdio.h>
#include <conio.h>
#include <dos.h>

int main(void)
{
    int value = 0;

    printf("The current status of your keyboard is:\n");
    value = peek(0x0040, 0x0017);
    if (value & 1)
       printf("Right shift on\n");
    else
       printf("Right shift off\n");

    if (value & 2)
       printf("Left shift on\n");
    else
       printf("Left shift off\n");

    if (value & 4)
       printf("Control key on\n");
    else
       printf("Control key off\n");

    if (value & 8)
       printf("Alt key on\n");
    else
       printf("Alt key off\n");

    if (value & 16)
       printf("Scroll lock on\n");
    else
       printf("Scroll lock off\n");

    if (value & 32)
       printf("Num lock on\n");
    else
       printf("Num lock off\n");

    if (value & 64)
       printf("Caps lock on\n");
    else
       printf("Caps lock off\n");

    return 0;
}
```

/* perror example */

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("perror.dat", "r");
    if (!fp)
       perror("Unable to open file for reading");
    return 0;
}
```

/* poke example */

```
#include <dos.h>
#include <conio.h>

int main(void)
{
    clrscr();
    cprintf("Make sure the scroll lock key is off and press any key\r\n");
    getch();
    poke(0x0000,0x0417,16);
    cprintf("The scroll lock is now on\r\n");
    return 0;
}
```

/* polar example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
    double x = 3.1, y = 4.2;
    complex z = complex(x,y);
    cout << "z = " << z << "\n";
    cout << "  has real part = " << real(z) << "\n";
    cout << "  and imaginary real part = " << imag(z) << "\n";
    cout << "z has complex conjugate = " << conj(z) << "\n";

    double mag = sqrt(norm(z));
    double ang = arg(z);
    cout << "The polar form of z is:\n";
    cout << "   magnitude = " << mag << "\n";
    cout << "   angle (in radians) = " << ang << "\n";
    cout << "Reconstructing z from its polar form gives:\n";
    cout << "   z = " << polar(mag,ang) << "\n";
    return 0;
}
```

/* poly example */

```
#include <stdio.h>
#include <math.h>

/* polynomial:  x**3 - 2x**2 + 5x - 1 */

int main(void)
{
  double array[] = { -1.0, 5.0, -2.0, 1.0
};
  double result;

  result = poly(2.0, 3, array);
  printf("The polynomial: x**3 - 2.0x**2 + 5x - 1 at 2.0 is %lf\n",
  result);
  return 0;
}
```

/* pow example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x = 2.0, y = 3.0;

  printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
  return 0;
}
```

/* pow10 example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double p = 3.0;

  printf("Ten raised to %lf is %lf\n", p, pow10(p));
  return 0;
}
```

/* putch example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char ch = 0;

  printf("Input a string:");
  while ((ch != '\r'))
  {
    ch = getch();
    putch(ch);
  }
  return 0;
}
```

/* raise example */

```
#include <signal.h>

int main(void)
{
    int a, b;

    a = 10;
    b = 0;
    if (b == 0)
    /* preempt divide by zero error */
       raise(SIGFPE);
    a = a / b;
    return 0;
}
```

```
/* rand example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i;

    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
    return 0;
}
```

/* random example */

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/* prints a random number in the range 0 to 99 */
int main(void)
 {
    randomize();
    printf("Random number in the 0-99 range: %d\n", random (100));
    return 0;
}
```

/* randomize example */

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
    int i;

    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
        printf("%d\n", rand() % 100);
    return 0;
}
```

/* real example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
    double x = 3.1, y = 4.2;
    complex z = complex(x,y);
    cout << "z = " << z << "\n";
    cout << "  has real part = " << real(z) << "\n";
    cout << "  and imaginary real part = " << imag(z) << "\n";
    cout << "z has complex conjugate = " << conj(z) << "\n";
    return 0;
}
```

/* remove example */

```
#include <stdio.h>

int main(void)
{
    char file[80];

    /* prompt for file name to delete */
    printf("File to delete: ");
    gets(file);

    /* delete the file */
    if (remove(file) == 0)
       printf("Removed %s.\n",file);
    else
       perror("remove");

    return 0;
}
```

/* rename example */

```
#include <stdio.h>

int main(void)
{
    char oldname[80], newname[80];

    /* prompt for file to rename and new name */
    printf("File to rename: ");
    gets(oldname);
    printf("New name: ");
    gets(newname);

    /* Rename the file */
    if (rename(oldname, newname) == 0)
       printf("Renamed %s to %s.\n", oldname, newname);
    else
       perror("rename");

    return 0;
}
```

/* rewind example */

```
#include <stdio.h>
#include <dir.h>

int main(void)
{
    FILE *fp;
    char *fname = "TXXXXXX", *newname, first;

    newname = mktemp(fname);
    fp = fopen(newname,"w+");
    fprintf(fp,"abcdefghijklmnopqrstuvwxyz");
    rewind(fp);
    fscanf(fp,"%c",&first);
    printf("The first character is: %c\n",first);
    fclose(fp);
    remove(newname);

    return 0;
}
```

/* rewinddir example */

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
 }

 void main(int argc,char *argv[])
 {
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

/* rmtmp example */

```
#include <stdio.h>
#include <process.h>

void main()
{
   FILE *stream;
   int i;

   /* Create temporary files */
   for (i = 1; i <= 10; i++)
   {
     if ((stream = tmpfile()) == NULL)
       perror("Could not open temporary file\n");
     else
       printf("Temporary file %d created\n", i);
   }
   /* Remove temporary files */
   if (stream != NULL)
     printf("%d temporary files deleted\n", rmtmp());
}
```

/* _searchenv example */

```
#include <stdio.h>
#include <stdlib.h>

char buf[_MAX_PATH];

int main(void)
{
  /* looks for TLINK */
  _searchenv("TLINK.EXE","PATH",buf);
  if (buf[0] == '\0')
    printf("TLINK.EXE not found\n");
  else
    printf("TLINK.EXE found in %s\n", buf);

  /* looks for non-existent file */
  _searchenv("NOTEXIST.FIL","PATH",buf);
  if (buf[0] == '\0')
    printf("NOTEXIST.FIL not found\n");
  else
    printf("NOTEXIST.FIL found in %s\n", buf);
  return 0;
}

  /* Program output

      TLINK.EXE found in C:\BIN\TLINK.EXE
      NOTEXIST.FIL not found
  */
```

/* searchpath example */

```
#include <stdio.h>
#include <dir.h>

int main(void)
{
    char *p;

    /* Looks for TLINK and returns a pointer
       to the path  */
    p = searchpath("TLINK.EXE");
    printf("Search for TLINK.EXE : %s\n", p);

    /* Looks for non-existent file  */
    p = searchpath("NOTEXIST.FIL");
    printf("Search for NOTEXIST.FIL : %s\n", p);

    return 0;
}
```

/* abort example */

```
#include <stdio.h>
#include <stdlib.h>

 int main(void)
 {
   printf("Calling abort()\n");
   abort();
   return 0; /* This is never reached */
 }
```

/* access example */

```
#include <stdio.h>
#include <io.h>

int file_exists(char *filename);

int main(void)
{
 printf("Does NOTEXIST.FIL exist: %s\n",
           file_exists("NOTEXISTS.FIL") ? "YES" : "NO");
 return 0;
}

int file_exists(char *filename)
{
  return (access(filename, 0) == 0);
}
```

/* arg example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
   double x = 3.1, y = 4.2;
   complex z = complex(x,y);
   cout << "z = " << z << "\n";
   cout << "  has real part = " << real(z) << "\n";
   cout << "  and imaginary part = " << imag(z) << "\n";
   cout << "z has complex conjugate = " << conj(z) << "\n";

   double mag = sqrt(norm(z));
   double ang = arg(z);
   cout << "The polar form of z is:\n";
   cout << "   magnitude = " << mag << "\n";
   cout << "   angle (in radians) = " << ang << "\n";
   cout << "Reconstructing z from its polar form gives:\n";
   cout << "   z = " << polar(mag,ang) << "\n";
   return 0;
}
```

/* assert example */

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

struct ITEM {
   int key;
   int value;
};

/* add item to list, make sure list is not  null */
void additem(struct ITEM *itemptr) {
   assert(itemptr != NULL);
   /* add item to list */
}

int main(void)
{
   additem(NULL);
   return 0;
}
```

/* atexit example */

```
#include <stdio.h>
#include <stdlib.h>

void exit_fn1(void)
{
   printf("Exit function #1 called\n");
}

void exit_fn2(void)
{
   printf("Exit function #2 called\n");
}

int main(void)
{
   /* post exit function #1 */
   atexit(exit_fn1);
   /* post exit function #2 */
   atexit(exit_fn2);
   return 0;
}
```

/* atof example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   float f;
   char *str = "12345.67";

   f = atof(str);
   printf("string = %s float = %f\n", str, f);
   return 0;
}
```

/* atoi example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int n;
   char *str = "12345.67";

   n = atoi(str);
   printf("string = %s integer = %d\n", str, n);
   return 0;
}
```

/* atol example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   long l;
   char *lstr = "98765432";

   l = atol(lstr);
   printf("string = %s integer = %ld\n", lstr, l);
   return(0);
}
```

/* bcd example */

```
#include <iostream.h>
#include <bcd.h>

double x = 10000.0;          // ten thousand dollars
bcd a = bcd(x/3,2);          // a third, rounded to nearest penny

int main(void)
{
   cout << "share of fortune = $" << a << "\n";
   return 0;
}
```

/* bdos example */

```
#include <stdio.h>
#include <dos.h>

/* Get current drive as 'A', 'B', ... */
char current_drive(void)
{
   char curdrive;

   /* Get current disk as 0, 1, ... */
   curdrive = bdos(0x19, 0, 0);
   return('A' + curdrive);
}

int main(void)
{
   printf("The current drive is %c:\n", current_drive());
   return 0;
}
```

/* bdosptr example */

```
#include <string.h>
#include <stdio.h>
#include <dir.h>
#include <dos.h>
#include <errno.h>
#include <stdlib.h>

#define  BUFLEN  80

int main(void)
{
   char  buffer[BUFLEN];
   int   test;

   printf("Enter full pathname of a directory\n");
   gets(buffer);

   test = bdosptr(0x3B,buffer,0);
      if(test)
      {
         printf("DOS error message: %d\n", errno);
         /* See errno.h for error listings */
         exit (1);
      }

   getcwd(buffer, BUFLEN);
   printf("The current directory is: %s\n", buffer);

   return 0;
}
```

/* calloc example */

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>

int main(void)
{
   char *str = NULL;

   /* allocate memory for string */
   str = (char *) calloc(10, sizeof(char));

   /* copy "Hello" into string */
   strcpy(str, "Hello");

   /* display string */
   printf("String is %s\n", str);

   /* free memory */
   free(str);

   return 0;
}
```

/* ceil and floor example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double number = 123.54;
   double down, up;

   down = floor(number);
   up = ceil(number);

   printf("original number     %5.2lf\n", number);
   printf("number rounded down %5.2lf\n", down);
   printf("number rounded up   %5.2lf\n", up);

   return 0;
}
```

/* cgets example */

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char buffer[83];
   char *p;

   /* There's space for 80 characters plus the NULL terminator */
   buffer[0] = 81;

   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);

   /* Leave room for 5 characters plus the NULL terminator */
   buffer[0] = 6;

   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
   return 0;
}
```

```
/* _chain_intr example */
#include <dos.h>
#include <stdio.h>
#include <process.h>

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

typedef void interrupt (*fptr)(__CPPARGS);

static void mesg(char *s)
{
  while (*s)
    bdos(2,*s++,0);
}

#pragma argsused
void interrupt handler2(unsigned bp, unsigned di)
{
  _enable();
  mesg("In handler 2.\r\n");
  if (di == 1)
    mesg("DI is 1\r\n");
  else
    mesg("DI is not 1\r\n");
  di++;
}

#pragma argsused
void interrupt handler1(unsigned bp, unsigned di)
{
  _enable();
  mesg("In handler 1.\r\n");
  if (di == 0)
    mesg("DI is 0\r\n");
  else
    mesg("DI is not 0\r\n");
  di++;
  mesg("Chaining to handler 2.\r\n");
  _chain_intr((fptr) handler2);
}

int main()
{
  _dos_setvect(128,(fptr) handler1);
  printf("About to generate interrupt 128\n");
  _DI = 0;
  geninterrupt(128);
  printf("DI was 0 before interrupt, is now 0x%x\n",_DI);
  return 0;
}
```

```
/* chdir example */
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>

char old_dir[MAXDIR];
char new_dir[MAXDIR];

int main(void)
{
   if (getcurdir(0, old_dir))
   {
      perror("getcurdir()");
      exit(1);
   }
   printf("Current directory is: \\%s\n", old_dir);

   if (chdir("\\"))
   {
      perror("chdir()");
      exit(1);
   }

   if (getcurdir(0, new_dir))
   {
      perror("getcurdir()");
      exit(1);
   }
   printf("Current directory is now: \\%s\n", new_dir);

   printf("\nChanging back to original directory: \\%s\n", old_dir);
   if (chdir(old_dir))
   {
      perror("chdir()");
      exit(1);
   }

   return 0;
}
```

/* chmod example */

```
/* NEW chmod() example: */

#include <errno.h>
#include <stdio.h>
#include <io.h>
#include <process.h>
#include <sys\stat.h>

void main(void)
{
  char filename[64];
  struct stat stbuf;
  int amode;

  printf("Enter name of file: ");
  scanf("%s", filename);
  if (stat(filename, &stbuf) != 0)
  {
    perror("Unable to get file information");
    exit(1);
  }
  if (stbuf.st_mode & S_IWRITE)
  {
    printf("Changing to read-only\n");
    amode = S_IREAD;
  }
  else
  {
    printf("Changing to read-write\n");
    amode = S_IREAD|S_IWRITE;
  }
  if (chmod(filename, amode) != 0)
  {
    perror("Unable to change file mode");
    exit(1);
  }
  exit(0);
}
```

/* chsize example */

```
#include <string.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char buf[11] = "0123456789";

   /* create text file containing 10 bytes */
   handle = open("DUMMY.FIL", O_CREAT);
   write(handle, buf, strlen(buf));

   /* truncate the file to 5 bytes in size */
   chsize(handle, 5);

   /* close the file */
   close(handle);
   return 0;
}
```

/* _clear87 and _status87 example */

```
#include <stdio.h>
#include <float.h>

int main(void)
{
   float x;
   double y = 1.5e-100;

   printf("\nStatus 87 before error: %X\n", _status87());

   x = y; /* create underflow and precision loss */
   printf("Status 87 after  error: %X\n", _status87());

   _clear87();
   printf("Status 87 after  clear: %X\n", _status87());

   y = x;

   return 0;
}
```

/* clearerr example */

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char ch;

   /* open a file for writing */
   fp = fopen("DUMMY.FIL", "w");

   /* force an error condition by attempting to read */
   ch = fgetc(fp);
   printf("%c\n",ch);

   if (ferror(fp))
   {
      /* display an error message */
      printf("Error reading from DUMMY.FIL\n");

      /* reset the error and EOF indicators */
      clearerr(fp);
   }

   fclose(fp);
   return 0;
}
```

/* clock example */

```
#include <time.h>
#include <stdio.h>
#include <dos.h>

int main(void)
{
   clock_t start, end;
   start = clock();

   delay(2000);

   end = clock();
   printf("The time was: %f\n", (end - start) / CLK_TCK);

   return 0;
}
```

/* clreol example */

```
#include <conio.h>

int main(void)

{
   clrscr();
   cprintf("The function CLREOL clears all characters from the\r\n");
   cprintf("cursor position to the end of the line within the\r\n");
   cprintf("current text window, without moving the cursor.\r\n");
   cprintf("Press any key to continue . . .");
   gotoxy(14, 4);
   getch();

   clreol();
   getch();

   return 0;
}
```

/* clrscr example */

```
#include <conio.h>

int main(void)
{
   int i;

   clrscr();
   for (i = 0; i < 20; i++)
      cprintf("%d\r\n", i);
   cprintf("\r\nPress any key to clear screen");
   getch();

   clrscr();
   cprintf("The screen has been cleared!");
   getch();

   return 0;
}
```

/* complex example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
   double x = 3.1, y = 4.2;
   complex z = complex(x,y);
   cout << "z = "<< z << "\n";
   cout << " and imaginary real part = " << imag(z) << "\n";
   cout << "z has complex conjugate =  " << conj(z) << " \n";
   return 0;
}
```

/* conj example */

```
#include <iostream.h>
#include <complex.h>

int main(void)
{
   double x = 3.1, y = 4.2;
   complex z = complex(x,y);
   cout << "z = "<< z << "\n";
   cout << " and imaginary real part = " << imag(z) << "\n";
   cout << "z has complex conjugate =  " << conj(z) << " \n";
   return 0;
}
```

```
/* country example */
#include <dos.h>
#include <stdio.h>

#define USA 0

int main(void)
{
   struct COUNTRY country_info;

   country(USA, &country_info);
   printf("The currency symbol for the USA is: %s\n",
           country_info.co_curr);
   return 0;
}
```

/* cputs example */

```
#include <conio.h>

int main(void)
{
   /* clear the screen */
   clrscr();

   /* create a text window */
   window(10, 10, 80, 25);

   /* output some text in the window */
   cputs("This is within the window\r\n");

   /* wait for a key */
   getch();
   return 0;
}
```

/* creattemp example */

```
#include <string.h>
#include <stdio.h>
#include <io.h>

int main(void)
{
   int handle;
   char pathname[128];

   strcpy(pathname, "\\");

   /* create a unique file in the root directory */
   handle = creattemp(pathname, 0);

   printf("%s was the unique file created.\n", pathname);
   close(handle);
   return 0;
}
```

```
/* ctrlbrk example */
#include <stdio.h>
#include <dos.h>

#define ABORT 0

int c_break(void)
{
   printf("Control-Break pressed.  Program aborting ...\n");
  return (ABORT);
}

void main(void)
{
   ctrlbrk(c_break);
   for(;;)
   {
      printf("Looping... Press <Ctrl-Break> to quit:\n");
   }
}
```

/* delline example */

```
#include <conio.h>

int main(void)
{
   clrscr();
   cprintf("The function DELLINE deletes the line containing the\r\n");
   cprintf("cursor and moves all lines below it one line up.\r\n");
   cprintf("DELLINE operates within the currently active text\r\n");
   cprintf("window.  Press any key to continue . . .");
   gotoxy(1,2);  /* Move the cursor to the second line and first column */
   getch();

   delline();
   getch();

   return 0;
}
```

/* difftime example */

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>

int main(void)
{
   time_t first, second;

   clrscr();
   first = time(NULL);  /* Gets system
                           time */
   delay(2000);         /* Waits 2 secs */
   second = time(NULL); /* Gets system time
                           again */

   printf("The difference is: %f seconds\n",difftime(second,first));
   getch();

   return 0;
}
```

/* dosexterr example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   FILE *fp;
   struct DOSERROR info;

   fp = fopen("perror.dat","r");
   if (!fp) perror("Unable to open file for reading");
   dosexterr(&info);

   printf("Extended DOS error information:\n");
   printf("   Extended error: %d\n",info.de_exterror);
   printf("            Class: %x\n",info.de_class);
   printf("           Action: %x\n",info.de_action);
   printf("      Error Locus: %x\n",info.de_locus);

   return 0;
}
```

/* dostounix example */

```
#include <time.h>
#include <stddef.h>
#include <dos.h>
#include <stdio.h>

int main(void)
{
  time_t t;
  struct time d_time;
  struct date d_date;
  struct tm *local;

  getdate(&d_date);
  gettime(&d_time);

  t = dostounix(&d_date, &d_time);
  local = localtime(&t);
  printf("Time and Date: %s\n", asctime(local));

  return 0;
}
```

/* __emit__ example */

```
#include <dos.h>

int main(void)
{
/* Emit code that will generate a print screen via int 5 */
   __emit__(0xcd,0x05);
   return 0;
}
```

/* eof example */

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char msg[] = "This is a test";
   char ch;

   /* create a file */
   handle = open("DUMMY.FIL",
                 O_CREAT | O_RDWR,
                 S_IREAD | S_IWRITE);

   /* write some data to the file */
   write(handle, msg, strlen(msg));

   /* seek to the beginning of the file */
   lseek(handle, 0L, SEEK_SET);

   /* reads chars from the file until it reaches EOF */
   do
   {
      read(handle, &ch, 1);
      printf("%c", ch);
   } while (!eof(handle));

   close(handle);
   return 0;
}
```

/* exp and expl example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double result;
   double x = 4.0;

   result = exp(x);
   printf("'e' raised to the power \
          of %lf (e ^ %lf) = %lf\n",
          x, x, result);

   return 0;
}
```

/* farcalloc example */

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
   char far *fptr;
   char *str = "Hello";

   /* allocate memory for the far pointer */
   fptr = (char far *) farcalloc(10, sizeof(char));

   /* copy "Hello" into allocated memory */
   /*
      Note: movedata is used because you might be in a small data model, in
      which case a normal string copy routine can not be used since it
      assumes the pointer size is near.
   */
   movedata(FP_SEG(str), FP_OFF(str),
      FP_SEG(fptr), FP_OFF(fptr),
      strlen(str));

   /* display string (note the F modifier) */
   printf("Far string is: %Fs\n", fptr);

   /* free the memory */
   farfree(fptr);

   return 0;
}
```

/* farmalloc example */

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
   char far *fptr;
   char *str = "Hello";

   /* allocate memory for the far pointer */
   fptr = (char far *) farmalloc(10);

   /* copy "Hello" into allocated memory */
   /*
    Note: movedata is used because we might be in a small data model,
      in which case a normal string copy routine can not be used since it
      assumes the pointer size is near.
   */
   movedata(FP_SEG(str), FP_OFF(str),
            FP_SEG(fptr), FP_OFF(fptr),
            strlen(str) + 1);

   /* display string (note the F modifier)
*/
   printf("Far string is: %Fs\n", fptr);

   /* free the memory */
   farfree(fptr);

   return 0;
}
```

/* fclose example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char buf[11] = "0123456789";

   /* create a file containing 10 bytes */
   fp = fopen("DUMMY.FIL", "w");
   fwrite(&buf, strlen(buf), 1, fp);

   /* close the file */
   fclose(fp);
   return 0;
}
```

/* fcloseall example */

```
#include <stdio.h>

int main(void)
{
   int streams_closed;

   /* open two streams */
   fopen("DUMMY.ONE", "w");
   fopen("DUMMY.TWO", "w");

   /* close the open streams */
   streams_closed = fcloseall();

   if (streams_closed == EOF)
      /* issue an error message */
      perror("Error");
   else
      /* print result of fcloseall() function */
      printf("%d streams were closed.\n", streams_closed);

   return 0;
}
```

/* feof example */

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* open a file for reading */
   stream = fopen("DUMMY.FIL", "r");

   /* read a character from the file */
   fgetc(stream);

   /* check for EOF */
   if (feof(stream))
      printf("We have reached end-of-file\n");

   /* close the file */
   fclose(stream);
   return 0;
}
```

/* ferror example */

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* open a file for writing */
   stream = fopen("DUMMY.FIL", "w");

   /* force an error condition by attempting to read */
   (void) getc(stream);

   if (ferror(stream))  /* test for an error on the stream */
   {
      /* display an error message */
      printf("Error reading from DUMMY.FIL\n");

      /* reset the error and EOF indicators */
      clearerr(stream);
   }

   fclose(stream);
   return 0;
}
```

/* fflush example */

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <io.h>

void flush(FILE *stream);

int main(void)
{
   FILE *stream;
   char msg[] = "This is a test";

   /* create a file */
   stream = fopen("DUMMY.FIL", "w");

   /* write some data to the file */
   fwrite(msg, strlen(msg), 1, stream);

   clrscr();
   printf("Press any key to flush DUMMY.FIL:");
   getch();

   /* flush the data to DUMMY.FIL without closing it */
   flush(stream);

   printf("\nFile was flushed, Press any key to quit:");
   getch();
   return 0;
}

void flush(FILE *stream)
{
     int duphandle;

     /* flush the stream's internal buffer */
     fflush(stream);

     /* make a duplicate file handle */
     duphandle = dup(fileno(stream));

     /* close the duplicate handle to flush the DOS buffer */
     close(duphandle);
}
```

/* fgetpos and fsetpos example */

```
#include <stdlib.h>
#include <stdio.h>

void showpos(FILE *stream);

int main(void)
{
   FILE *stream;
   fpos_t filepos;

   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");

   /* save the file pointer position */
   fgetpos(stream, &filepos);

   /* write some data to the file */
   fprintf(stream, "This is a test");

   /* show the current file position */
   showpos(stream);

   /* set a new file position, display it */
   if (fsetpos(stream, &filepos) == 0)
     showpos(stream);
   else
   {
      fprintf(stderr, "Error setting file pointer.\n");
      exit(1);
   }

   /* close the file */
   fclose(stream);
   return 0;
}

void showpos(FILE *stream)
{
   fpos_t pos;

   /* display the current file pointer
      position of a stream */
   fgetpos(stream, &pos);
   printf("File position: %ld\n", pos);
}
```

/* filelength example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char buf[11] = "0123456789";

   /* create a file containing 10 bytes */
   handle = open("DUMMY.FIL", O_CREAT);
   write(handle, buf, strlen(buf));

   /* display the size of the file */
   printf("file length in bytes: %ld\n", filelength(handle));

   /* close the file */
   close(handle);
   return 0;
}
```

/* fileno example */

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int handle;

   /* create a file */
   stream = fopen("DUMMY.FIL", "w");

   /* obtain the file handle associated with the stream */
   handle = fileno(stream);

   /* display the handle number */
   printf("handle number: %d\n", handle);

   /* close the file */
   fclose(stream);
   return 0;
}
```

/* flushall example */

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* create a file */
   stream = fopen("DUMMY.FIL", "w");

   /* flush all open streams */
   printf("%d streams were flushed.\n", flushall());

   /* close the file */
   fclose(stream);
   return 0;
}
```

/* fmod and fmodl example */

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double x = 5.0, y = 2.0;
   double result;

   result = fmod(x,y);
   printf("The remainder of (%lf / %lf) is %lf\n", x, y, result);
   return 0;
}
```

/* FP_OFF, and FP_SEG example*/

```
#include <stdio.h>
#include <dos.h>

main()
{
  char *str = "Hello\n";

  printf("The address pointed to by str is %04X:%04X\n",
          FP_SEG(str), FP_OFF(str));
  printf("The address of str is %04X:%04X\n", FP_SEG(&str), FP_OFF(&str));
  return 0;
}
```

/* fread example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char msg[] = "this is a test";
   char buf[20];

   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }

   /* write some data to the file */
   fwrite(msg, strlen(msg)+1, 1, stream);

   /* seek to the beginning of the file */
   fseek(stream, SEEK_SET, 0);

   /* read the data and display it */
   fread(buf, strlen(msg)+1, 1, stream);
   printf("%s\n", buf);

   fclose(stream);
   return 0;
}
```

/* frexp and frexpl examples */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double mantissa, number;
   int exponent;

   number = 8.0;
   mantissa = frexp(number, &exponent);

   printf("The number %lf is ", number);
   printf("%lf times two to the ", mantissa);
   printf("power of %d\n", exponent);

   return 0;
}
```

```
/* fseek example */
#include <stdio.h>

long filesize(FILE *stream);

int main(void)
{
   FILE *stream;

   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("Filesize of MYFILE.TXT is %ld bytes\n", filesize(stream));
   fclose(stream);
   return 0;
}

long filesize(FILE *stream)
{
   long curpos, length;

   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}
```

/* ftell example */

```
#include <stdio.h>
int main(void)
{
   FILE *stream;

   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("The file pointer is at byte %ld\n", ftell(stream));
   fclose(stream);
   return 0;
}
```

/* ftime example */

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys\timeb.h>

/* pacific standard & daylight savings */
char *tzstr = "TZ=PST8PDT";

int main(void)
{
   struct timeb t;
   putenv(tzstr);
   tzset();
   ftime(&t);
   printf("Seconds since 1/1/1970 GMT: %ld\n", t.time);
   printf("Thousandths of a second: %d\n", t.millitm);
   printf("Difference between local time and GMT: %d\n", t._timezone);
   printf("Daylight savings in effect (1) not (0): %d\n", t.dstflag);
   return 0;
}
```

/* _fullpath example */

```
#include <stdio.h>
#include <stdlib.h>

char buf[_MAX_PATH];

void main(int argc, char *argv[])
{
  for ( ; argc; argv++, argc--)
  {
    if (_fullpath(buf, argv[0], _MAX_PATH) == NULL)
      printf("Unable to obtain full path of %s\n",argv[0]);
    else
      printf("Full path of %s is %s\n",argv[0],buf);
  }
}
```

/* fwrite example */

```
#include <stdio.h>

struct mystruct
{
  int i;
  char ch;
};

int main(void)
{
   FILE *stream;
   struct mystruct s;

   if ((stream = fopen("TEST.$$$", "wb")) == NULL) /* open file TEST.$$$ */
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   s.i = 0;
   s.ch = 'A';
   fwrite(&s, sizeof(s), 1, stream); /* write struct s to file */
   fclose(stream); /* close file */
   return 0;
}
```

/* gcvt example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char str[25];
   double num;
   int sig = 5; /* significant digits */

   /* a regular number */
   num = 9.876;
   gcvt(num, sig, str);
   printf("string = %s\n", str);

   /* a negative number */
   num = -123.4567;
   gcvt(num, sig, str);
   printf("string = %s\n", str);

   /* scientific notation */
   num = 0.678e5;
   gcvt(num, sig, str);
   printf("string = %s\n", str);

   return(0);
}
```

/* geninterrupt example */

```
#include <conio.h>
#include <dos.h>

/* function prototype */
void writechar(char ch);

int main(void)
{
   clrscr();
   gotoxy(80,25);
   writechar('*');
   getch();
   return 0;
}

/*
   outputs a character at the current cursor
   position using the video BIOS to avoid
   the scrolling of the screen when writing
   to location (80,25).
*/

void writechar(char ch)
{
   struct text_info ti;
   /* grab current text settings */
   gettextinfo(&ti);
   /* interrupt 0x10 sub-function 9 */
   _AH = 9;
   /* character to be output */
   _AL = ch;
   _BH = 0;                    /* video page */
   _BL = ti.attribute;  /* video attribute */
   _CX = 1;            /* repetition factor */
   geninterrupt(0x10);  /* output the char */
}
```

/* getcbrk and setcbrk example */

```
#include <dos.h>
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int break_flag;

   printf("Enter 0 to turn control break off\n");
   printf("Enter 1 to turn control break on\n");

   break_flag = getch() - 0;

   setcbrk(break_flag);

   if (getcbrk())
      printf("Cntrl-brk flag is on\n");
   else
      printf("Cntrl-brk flag is off\n");
   return 0;
}
```

/* segread example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   struct SREGS segs;

   segread(&segs);
   printf("Current segment register settings\n\n");
   printf("CS: %X   DS: %X\n", segs.cs, segs.ds);
   printf("ES: %X   SS: %X\n", segs.es, segs.ss);

   return 0;
}
```

/* setmem example */

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>

int main(void)
{
   char *dest;

   dest = (char *) calloc(21, sizeof(char));
   setmem(dest, 20, 'c');
   printf("%s\n", dest);

   return 0;
}
```

```
/* setmode example */
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
int main (int argc, char ** argv )
(
  FILE *fp;
  int newmode;
  long where;
  char buf[256];

  fp = fopen( argv[1], "r+" );
  if ( !fp )
  {
    printf( "Couldn't open %s\n", argv[1] );
    return -1;
  }

  newmode = setmode( fileno( fp ), O_BINARY );
  if ( newmode == -1 )
  {
    printf( "Coudn't set mode of %s\n", argv[1] );
    return -2
  }

  fp->flags |= _F_BIN;
  where = ftell( fp );
  printf ("file position: %d\n", where );
  fread( buf, 1, 1, fp );
  where = ftell ( fp );
  printf( "read %c, file position: %ld\n", *buf, where );
  fclose ( fp );
  return 0;
}
```

/* signal example */

```
/* signal example */

/*
    This example installs a signal handler routine for SIGFPE,
    catches an integer overflow condition, makes an adjustment to AX
    register, and returns.  This example program MAY cause your computer
    to crash, and will produce runtime errors depending on which memory
    model is used.
*/

#pragma inline
#include <stdio.h>
#include <signal.h>

#ifdef __cplusplus
   typedef void (*fptr)(int);
#else
   typedef void (*fptr)();
#endif

void Catcher(int *reglist)
{
   signal(SIGFPE, (fptr)Catcher);  //  ******reinstall signal handler

   printf("Caught it!\n"); *(reglist + 8) = 3; /* make return AX = 3 */
}

int main(void)
{
   signal(SIGFPE, (fptr)Catcher);  /* cast Catcher to appropriate type */

   asm     mov     ax,07FFFH       /* AX = 32767 */
   asm     inc     ax              /* cause overflow */
   asm     into                    /* activate handler */

   /* The handler set AX to 3 on return. If that had not happened,
      there would have been another exception when the next 'into'
      executed after the 'dec' instruction. */

   asm     dec     ax              /* no overflow now */
   asm     into                    /* doesn't activate */
   return 0;
}
```

/* sleep example */

```
#include <dos.h>
#include <stdio.h>

int main(void)
{
   int i;

   for (i=1; i<5; i++)
   {
      printf("Sleeping for %d seconds\n", i);
      sleep(i);
   }
   return 0;
}
```

/* sqrt example */

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 4.0, result;

   result = sqrt(x);
   printf("The square root of %lf is %lf\n", x, result);
   return 0;
}
```

/* srand example */

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
   int i;
   time_t t;

   srand((unsigned) time(&t));
   printf("Ten random numbers from 0 to 99\n\n");
   for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
   return 0;
}
```

/* stpcpy example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";

   stpcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

/*strcat example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char destination[25];
   char *blank = " ", *c = "C++", *Borland = "Borland";

   strcpy(destination, Borland);
   strcat(destination, blank);
   strcat(destination, c);

   printf("%s\n", destination);
   return 0;
}
```

/* strchr example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char string[15];
   char *ptr, c = 'r';

   strcpy(string, "This is a string");
   ptr = strchr(string, c);
   if (ptr)
      printf("The character %c is at position: %d\n", c, ptr-string);
   else
      printf("The character was not found\n");
   return 0;
}
```

/* strcoll example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *two = "International";
   char *one = "Borland";
   int check;

   check = strcoll(one, two);
   if (check == 0)
      printf("The strings are equal\n");
   if (check <  0)
      printf("%s comes before %s\n", one, two);
   if (check >  0)
      printf("%s comes before %s\n", two, one);
   return 0;
}
```

/* strcpy example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";

   strcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

/* _strdate example */

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];

  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

/* strdup example */

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main(void)
{
   char *dup_str, *string = "abcde";

   dup_str = strdup(string);
   printf("%s\n", dup_str);
   free(dup_str);

   return 0;
}
```

/* strftime example */

```
#include <stdio.h>
#include <time.h>
#include <dos.h>

int main(void)
{
   struct tm *time_now;
   time_t secs_now;
   char str[80];

   tzset();
   time(&secs_now);
   time_now = localtime(&secs_now);
   strftime(str, 80,
            "It is %M minutes after %I o'clock (%Z)  %A, %B %d 19%y",
            time_now);
   printf("%s\n",str);
   return 0;
}
```

/*strlen example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "Borland International";

   printf("%d\n", strlen(string));
   return 0;
}
```

/*strncat example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char destination[25];
   char *source = " States";

   strcpy(destination, "United");
   strncat(destination, source, 7);
   printf("%s\n", destination);
   return 0;
}
```

/* strncpy example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";

   strncpy(string, str1, 3);
   string[3] = '\0';
   printf("%s\n", string);
   return 0;
}
```

/* strnset example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz";
   char letter = 'x';

   printf("string before strnset: %s\n", string);
   strnset(string, letter, 13);
   printf("string after  strnset: %s\n", string);

   return 0;
}
```

/* strpbrk example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "onm";
   char *ptr;

   ptr = strpbrk(string1, string2);

   if (ptr)
      printf("strpbrk found first character: %c\n", *ptr);
   else
      printf("strpbrk didn't find character in set\n");

   return 0;
}
```

/* strrchr example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char string[15];
   char *ptr, c = 'r';

   strcpy(string, "This is a string");
   ptr = strrchr(string, c);
   if (ptr)
      printf("The character %c is at position: %d\n", c, ptr-string);
   else
      printf("The character was not found\n");
   return 0;
}
```

/* strrev example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *forward = "string";

   printf("Before strrev(): %s\n", forward);
   strrev(forward);
   printf("After strrev():  %s\n", forward);
   return 0;
}
```

/* strset example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char string[10] = "123456789";
   char symbol = 'c';

   printf("Before strset(): %s\n", string);
   strset(string, symbol);
   printf("After strset():  %s\n", string);
   return 0;
}
```

/* strstr example */

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *str1 = "Borland International", *str2 = "nation", *ptr;

   ptr = strstr(str1, str2);
   printf("The substring is: %s\n", ptr);
   return 0;
}
```

/* _strtime example */

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];

  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

/* strtok example */

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char input[16] = "abc,d";
   char *p;

   /* strtok places a NULL terminator
   in front of the token, if found */
   p = strtok(input, ",");
   if (p)   printf("%s\n", p);

   /* A second call to strtok using a NULL
   as the first parameter returns a pointer
   to the character following the token  */
   p = strtok(NULL, ",");
   if (p)   printf("%s\n", p);
   return 0;
}
```

/* strxfrm example */

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main(void)
{
   char *target;
   char *source = "Frank Borland";
   int length;

   /* allocate space for the target string */
   target = (char *) calloc(80, sizeof(char));

   /* copy the source over to the target and get the length */
   length = strxfrm(target, source, 80);

   /* print out the results */
   printf("%s has the length %d\n", target, length);
   return 0;
}
```

/* swab example */

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[15] = "rFna koBlrna d";
char target[15];

int main(void)
{
   swab(source, target, strlen(source));
   printf("This is target: %s\n", target);
   return 0;
}
```

/* system example */

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   printf("About to spawn command.com and run a DOS command\n");
   system("dir");
   return 0;
}
```

/* tell example */

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
   int handle;
   char msg[] = "Hello world";

   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT | O_APPEND)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   printf("The file pointer is at byte %ld\n", tell(handle));
   close(handle);
   return 0;
}
```

/* tempnam example */

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  FILE *stream;
  int i;
  char *name;

  for (i = 1; i <= 10; i++) {
    if ((name = tempnam("\\tmp","wow")) == NULL)
      perror("tempnam couldn't create name");
    else {
      printf("Creating %s\n",name);
      if ((stream = fopen(name,"wb")) == NULL)
        perror("Could not open temporary file\n");
      else
        fclose(stream);
    }
    free(name);
  }
  printf("Warning: temp files not deleted.\n");
}
```

/* textmode example */

```
#include <conio.h>

int main(void)
{
   textmode(BW40);
   cprintf("ABC");
   getch();

   textmode(C40);
   cprintf("ABC");
   getch();

   textmode(BW80);
   cprintf("ABC");
   getch();

   textmode(C80);
   cprintf("ABC");
   getch();

   textmode(MONO);
   cprintf("ABC");
   getch();

   return 0;
}
```

/* tmpfile example */

```
#include <stdio.h>
#include <process.h>

int main(void)
{
   FILE *tempfp;

   tempfp = tmpfile();
   if (tempfp)
      printf("Temporary file created\n");
   else
   {
      printf("Unable to create temporary file\n");
      exit(1);
   }

   return 0;
}
```

/* tmpnam example */

```
#include <stdio.h>

int main(void)
{
   char name[13];

   tmpnam(name);
   printf("Temporary name: %s\n", name);
   return 0;
}
```

/* toascii example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int number, result;
   number = 511;
   result = toascii(number);
   printf("%d %d\n", number, result);
   return 0;
}
```

/* tzset example */

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   time_t td;

   putenv("TZ=PST8PDT");
   tzset();
   time(&td);
   printf("Current time = %s\n", asctime(localtime(&td)));
   return 0;
}
```

/* ungetc example */

```
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   int i=0;
   char ch;

   puts("Input an integer followed by a char:");

   /* read chars until non digit or EOF */
   while((ch = getchar()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */

   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetc(ch, stdin);

   printf("i = %d, next char in buffer = %c\n", i, getchar());
   return 0;
}
```

/* ungetch example */

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

int main( void )
{
   int i=0;
   char ch;

   puts("Input an integer followed by a char:");

   /* read chars until non digit or EOF */
   while((ch = getche()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */

   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetch(ch);

   printf("\n\ni = %d, next char in buffer = %c\n", i, getch());
   return 0;
}
```

/* unixtodos example */

```
#include <stdio.h>
#include <dos.h>

char *month[] = {"---", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

#define SECONDS_PER_DAY 86400L  /* the number of seconds in one day */

struct date dt;
struct time tm;

int main(void)
{
   unsigned long val;

/* get today's date and time */
   getdate(&dt);
   gettime(&tm);
   printf("today is %d %s %d\n", dt.da_day, month[dt.da_mon], dt.da_year);

/*convert date and time to unix format (num of seconds since Jan 1, 1970*/
   val = dostounix(&dt, &tm);
/* subtract 42 days worth of seconds */
   val -= (SECONDS_PER_DAY * 42);

/* convert back to dos time and date */
   unixtodos(val, &dt, &tm);
   printf("42 days ago it was %d %s %d\n",
          dt.da_day, month[dt.da_mon], dt.da_year);
   return 0;
}
```

/* unlink example */

```
#include <stdio.h>
#include <io.h>

int main(void)
{
   FILE *fp = fopen("junk.jnk","w");
   int status;

   fprintf(fp,"junk");

   status = access("junk.jnk",0);
   if (status == 0)
      printf("File exists\n");
   else
      printf("File doesn't exist\n");

   fclose(fp);
   unlink("junk.jnk");
   status = access("junk.jnk",0);
   if (status == 0)
      printf("File exists\n");
   else
      printf("File doesn't exist\n");

   return 0;
}
```

/* umask example */

```
#include <io.h>
#include <stdio.h>
#include <sys\stat.h>

#define FILENAME "TEST.$$$"

int main(void)
{
    unsigned oldmask;

    FILE *f;
    struct stat statbuf;

    /* Cause subsequent files to be created as read-only */
    oldmask = umask(S_IWRITE);
    printf("Old mask = 0x%x\n",oldmask);

    /* Create a zero-length file */
    if ((f = fopen(FILENAME,"w+")) == NULL)
    {
        perror("Unable to create output file");
        return (1);
    }
    fclose(f);

    /* Verify that the file is read-only */
    if (stat(FILENAME,&statbuf) != 0)
    {
        perror("Unable to get information about output file");
        return (1);
    }
    if (statbuf.st_mode & S_IWRITE)
        printf("Error! %s is writable!\n",FILENAME);
    else
        printf("Success! %s is not writable.\n",FILENAME);
    return (0);
}
```

/* utime example */

```
/* Copy timestamp from one file to another */

#include <sys\stat.h>
#include <utime.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  struct stat src_stat;
  struct utimbuf times;
  if(argc != 3) {
    printf( "Usage: copytime <source file> <dest file>\n" );
    return 1;
  }

  if (stat(argv[1],&src_stat) != 0) {
    perror("Unable to get status of source file");
    return 1;
  }

  times.modtime = times.actime = src_stat.st_mtime;
  if (utime(argv[2],&times) != 0) {
    perror("Unable to set time of destination file");
    return 1;
  }
  return 0;
}
```

/* va_arg example */

```
#include <stdio.h>
#include <stdarg.h>

/* calculate sum of a 0 terminated list */
void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;
   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0) {
      total += arg;
   }
   printf(msg, total);
   va_end(ap);
}

int main(void) {
   sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
   return 0;
}
```

/* wherex and wherey example */

```
#include <conio.h>

int main(void)
{
   clrscr();
   gotoxy(10,10);
   cprintf("Current location is X: %d  Y: %d\r\n", wherex(), wherey());
   getch();

   return 0;
}
```

/* window example */

```
#include <conio.h>

int main(void)
{

   window(10,10,40,11);
   textcolor(BLACK);
   textbackground(WHITE);
   cprintf("This is a test\r\n");

   return 0;
}
```

/* getpsp example */

```
#include <stdio.h>
#include <dos.h>

int main(void)
{
   static char command[128];
   char far *cp;
   int len, i;

   printf("The program segment prefix is: %u\n", getpsp());

/*
_psp is preset to segment of the PSP. Command line is located at offset
  0x81 from start of PSP
*/
   cp = (char *) MK_FP(_psp, 0x80);
   len = *cp;

   for (i = 0; i < len; i++)
      command[i] = cp[i+1];

   printf("Command line: %s\n", command);

   return 0;
}
```

/* stackavail example */

```
#include <malloc.h>
#include <stdio.h>

int main(void)
{
  char *buf;

  printf("\nThe stack: %u\tstack pointer: %u", stackavail(), _SP);
  buf = (char *) alloca(100 * sizeof(char));
  printf("\nNow, the stack: %u\tstack pointer: %u", stackavail(), _SP);
  return 0;
}

/* ***********************************************

     Program output

     The stack: 64046       stack pointer: 65524
     Now, the stack: 63946  stack pointer: 65424

   *********************************************** */
```

/* set_new_handler example */

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

void mem_warn() {
  cerr << "\nCan't allocate!";
  exit(1);
  }

void main(void) {
  set_new_handler(mem_warn);

  char *ptr = new char[100];
  cout << "\nFirst allocation: ptr = " << hex << long(ptr);
  ptr = new char[64000U];
  cout << "\nFinal allocation: ptr = " << hex << long(ptr);
  set_new_handler(0);  // Reset to default.
}
```

/* isalpha example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isalpha(c))
    printf("%c is alphabetical\n",c);
  else printf("%c is not alphabetical\n",c);

  return 0;
}
```

/* isalnum example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isalnum(c))
    printf("%c is alphanumeric\n",c);
  else printf("%c is not alphanumeric\n",c);

  return 0;
}
```

/* isascii example */

```
#include <stdio.h>
#include <ctype.h>
#include <stdio.h>
int main(void)
{
  char c = 'C';

  if (isascii(c))
    printf("%c is ascii\n",c);
  else printf("%c is not ascii\n",c);
  return 0;
}
```

/* iscntrl example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';
  if (iscntrl(c))
    printf("%c is a control character\n",c);
  else printf("%c is not a control character\n",c);

  return 0;
}
```

/* isdigit example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isdigit(c))
    printf("%c is a digit\n",c);
  else printf("%c is not a digit\n",c);

  return 0;
}
```

/* isgraph example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isgraph(c))
    printf("%c is a graphic character\n",c);
  else printf("%c is not a graphic character\n",c);

  return 0;
}
```

/* islower example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (islower(c))
    printf("%c is a lowercase character\n",c);
  else printf("%c is not a lowercase character\n",c);

  return 0;
}
```

/* isprint example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isprint(c))
    printf("%c is a printable character\n",c);
  else printf("%c is not a printable character\n",c);

  return 0;
}
```

/* ispunct example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (ispunct(c))
    printf("%c is a punctuation character\n",c);
  else printf("%c is not a punctuation character\n",c);

  return 0;
}
```

/* isspace example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isspace(c))
    printf("%c is white space\n",c);
  else printf("%c is not white space\n",c);

  return 0;
}
```

/* isupper example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isupper(c))
    printf("%c is an uppercase character\n",c);
  else printf("%c is not an uppercase character\n",c);

  return 0;
}
```

/* isxdigit example */

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char c = 'C';

  if (isxdigit(c))
    printf("%c is a hexidecimal digit\n",c);
  else printf("%c is not a hexidecimal digit\n",c);

  return 0;
}
```

/* mblen example */

```
#include <stdlib.h>
#include <stdio.h>

void main (void)
{
  int i ;
  char *mulbc = (char *)malloc( sizeof( char) );
  wchar_t widec = L'a';
  printf (" convert a wide character to multibyte character:\n" );
  i = wctomb (mulbc, widec);
  printf( "\tCharacters converted: %u\n", i);
  printf( "\tMultibyte character: %x\n\n", mulbc);

  printf( " Find length--in byte-- of multibyte character:\n");
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLenght--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);


  printf( " Attempt to find length of a Wide character Null:\n");
  widec = L'\0';
  wctomb(mulbc, widec);
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLenght--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);

}
```

/* mbstowcs example */

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int x;
  char    *mbst = (char *)malloc(MB_CUR_MAX);
  wchar_t *pwst = L"Hi";
  wchar_t *pwc      = (wchar_t *)malloc(sizeof( wchar_t));

  printf ("Convert to multibyte string:\n");
  x = wcstombs (mbst, pwst, MB_CUR_MAX);
  printf ("\tCharacters converted %u\n",x);
  printf ("\tHEx value of first");
  printf (" multibyte character: %#.4x\n\n", mbst);

  printf ("Convert back to wide character string:\n");
  x = mbstowcs(pwc, mbst, MB_CUR_MAX);
  printf( "\tCharacters converted: %u\n",x);
  printf( "\tHex value of first");
  printf( "wide character: %#.4x\n\n", pwc);
}
```

/* mbtowc example */

```
#include <stdlib.h>
#include<stdio.h>

void main(void)
{
  int     x;
  char     *mbchar    = (char *)calloc(1, sizeof( char));
  wchar_t  wchar      = L'a';
  wchar_t   *pwcnull = NULL;
  wchar_t  *pwchar    = (wchar_t *)calloc(1,  sizeof( wchar_t ));

  printf ("Convert a wide character to multibyte character:\n");
  x = wctomb( mbchar, wchar);
  printf( "\tCharacters converted: %u\n", x);
  printf( "\tMultibyte character: %x\n\n", mbchar);

  printf ("Convert multibyte character back to a wide character:\n");
  x = mbtowc( pwchar, mbchar, MB_CUR_MAX );
  printf( "\tBytes converted: %u\n", x);
  printf( "\tWide character: %x\n\n", pwchar);

  printf ("Atempt to convert when target is NULL\n" );
  printf (" returns the length of the multibyte character:\n" );
  x = mbtowc (pwcnull, mbchar, MB_CUR_MAX );
  printf ( "\tlength of multibyte character:%u\n\n", x );

  printf ("Attempt to convert a NULL pointer to a" );
  printf (" wide character:\n" );
  mbchar = NULL;
  x = mbtowc (pwchar, mbchar, MB_CUR_MAX);
  printf( "\tBytes converted: %u\n", x );
  }
```

/* MK_FP example */

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <dos.h>
#include <malloc.h>

main()
{

  char *str = "hello\n";
  char far *farstr;

  printf ("the address pointed to by str is %04X:%04X\n",
    FP_SEG(str), FP_OFF(str));
  farstr = (char far *)MK_FP( FP_SEG(str),  FP_OFF(str));

  printf ("the string pointed by far pointer is %s\n", farstr);
  return 0;
}
```

/* _msize example */

```
/* _msize works as a 32-bit command, not as a 16-bit command */
#include <malloc.h>         /* malloc() _msize() */
#include <stdio.h>             /* printf() */

int main( )
{
  int size;
  int *buffer;

  buffer = malloc(100 * sizeof(int));
  size = _msize(buffer);
  printf("Allocated %d bytes for 100 integers\n", size);

  return(0);
}
```

/* offsetof example */

```
/*
This program uses the offsetof command to show the effect
of changing alignment boundaries within a structure.
It produces this output:

  In STRUCT1, two_bytes begins at byte 1.
  In STRUCT2, two_bytes begins at byte 2.

By default, the 16-bit compiler aligns structure members
at 1-byte boundaries.  With the -a2 flag set, the compiler
aligns fields on even boundaries.

The CPU often processes structure elements more quickly
when they align on even boundaries.
*/

#include <stddef.h>      // offsetof()
#include <stdio.h>      // printf()

#pragma option -a1      // align on bytes (default)

typedef struct {
  char one_byte;
  int two_bytes;
} STRUCT1;


#pragma option -a2      // align on even bytes

typedef struct {
  char one_byte;
  int two_bytes;
} STRUCT2;

#pragma option -a.      // restore command-line option


void main()
{
  printf( "In STRUCT1, two_bytes begins at byte %d.\n",
    offsetof(STRUCT1, two_bytes) );

  printf( "In STRUCT2, two_bytes begins at byte %d.\n",
    offsetof(STRUCT2, two_bytes) );
}
```

/* _pipe example */

```
/* _pipe example */
#include <windows.h>   //for SECURITY_ATTRIBUTES
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>

void main(int argc, char *argv[])
{
    int retcode, stat, pid;
    char asc_handle[10];
    SECURITY_ATTRIBUTES sa;
    HANDLE s_hFileMap;
   LPVOID lpView;

    if (argc > 1)   /* this is the child */
    {
        /* Get the read pipe handle from command line,
         * and set the handle to binary.
         */

        printf("Child: The handle passed as 2nd argument is: %s\n",
  argv[1]);
        s_hFileMap = (HANDLE) atoi(argv[1]);

        lpView = MapViewOfFile(s_hFileMap,
                                    FILE_MAP_READ|FILE_MAP_WRITE,
                                    0,0,0);
        if (lpView == NULL)
        {
            perror("Child: unable to read pipe");
            retcode = 255;
        }

        printf("Child: returning %s to parent\n",lpView);

        retcode = atoi((char*) lpView);
        UnmapViewOfFile(lpView);
        CloseHandle(s_hFileMap);       //Child is responsible for this
        exit(retcode);
    }
    else            /* this is the parent */
    {
        //Here we set up the security attributes of the file mapping object
        //so that we can inherit it from the child process.  Alternatively
        //and more cheaply, we could use just the name of the mapping
  object
        //once inside the child process and call OpenFileMapping with that

        //name.
        sa.nLength = sizeof(sa);
        sa.lpSecurityDescriptor = NULL;
        sa.bInheritHandle = TRUE;
```

```
        s_hFileMap = CreateFileMapping((HANDLE) 0xFFFFFFFF,//in memory
                                       &sa,                //security
    attrib
                                       PAGE_READWRITE,
                                       0,                  //min. size
                                       256,                //size

                                       NULL);              //give mapping
    object no name

        lpView = MapViewOfFile(s_hFileMap,
                                      FILE_MAP_READ|FILE_MAP_WRITE,
                                      0,0,0);
        if (lpView == NULL)
        {
            perror("Parent: unable to create file mapping");
            exit(1);
        }

        sprintf(asc_handle,"%d",s_hFileMap);
        retcode = 10;
        if (sprintf((char*)lpView,"%d",retcode) == EOF)

        {
            perror("Parent: unable to write to pipe");
            exit(1);
        }

        /* Call ourself with read handle as argument.
         */
        if ((pid = spawnl(P_NOWAIT, argv[0], argv[0],
                          asc_handle, NULL)) == -1)
            perror("Parent: spawnl failed");
        else
        {
            printf("Parent: spawned child process %d\n",pid);
            if (wait(&stat) != pid)
                perror("Parent: wait failure");

            else
            {
                if ((stat & 0xff) == 0)
                    printf("Parent: child returned %d\n", stat >> 8);
                else
                    printf("Parent: child terminated abnormally\n");

            }
        }
        UnmapViewOfFile(lpView);
        CloseHandle(s_hFileMap);
        exit(0);
    }
}
```

/* send example */

```
/*
 There are two short programs here.  SEND spawns a child
 process, RECEIVE.  Each process holds one end of a
 pipe.  The parent transmits its command-line argument
 to the child, which prints the string and exits.

 IMPORTANT:  The parent process must be linked with
 the \32bit\fileinfo.obj file.  The code in fileinfo
 enables a parent to share handles with a child.
 Without this extra information, the child cannot use
 the handle it receives.
*/

/* SEND */

#include <fcntl.h>            // _pipe()
#include <io.h>             // write()
#include <process.h>          // spawnl() cwait()
#include <stdio.h>           // puts() perror()
#include <stdlib.h>           // itoa()
#include <string.h>           // strlen()

#define DECIMAL_RADIX 10      // for atoi()
enum PIPE_HANDLES { IN, OUT };  // to index the array of handles

int main(int argc, char *argv[])
{
  int handles[2];            // in- and
//outbound pipe handles
  char handleStr[10];         // a handle
//stored as a string
  int pid;
  // system's ID for child process

  if (argc <= 1)
  {
    puts("No message to send.");
    return(1);
  }

  if (_pipe(handles, 256, O_TEXT) != 0)
  {
    perror("Cannot create the pipe");
    return(1);
  }

  // store handle as a string for passing on the command line
  itoa(handles[IN], handleStr, DECIMAL_RADIX);

  // create the child process, passing it the inbound pipe handle
     spawnl(P_NOWAIT, "receive.exe", "receive.exe", handleStr, NULL);

  // transmit the message
  write(handles[OUT], argv[1], strlen(argv[1])+1);
```

```
  // when done with the pipe, close both handles
  close(handles[IN]);
  close(handles[OUT]);

  // wait for the child to finish
  wait(NULL);
  return(0);
}
```

/* _setcursortype example */

```
#include <conio.h>

int main( )
{
  // tell the user what to do
  clrscr();
  cputs("Press any key three times.\n\r");
  cputs("Each time the cursor will change shape.\n\r");

  gotoxy(1,5);           // show a solid cursor
  cputs("Now the cursor is solid.\n\r");
  _setcursortype(_SOLIDCURSOR);

  while(!kbhit()) {};         // wait to proceed
  getch();


  gotoxy(1,5);           // remove the cursor
  cputs("Now the cursor is gone.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NOCURSOR);

  while(!kbhit()) {};         // wait to proceed
  getch();


  gotoxy(1,5);                 // show a normal cursor
  cputs("Now the cursor is normal.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NORMALCURSOR);

  while(!kbhit()) {};         // wait to proceed
  getch();

  clrscr();
  return(0);
}
```

/* _dos_commit example */

```
#include <dos.h>
#include <errno.h>
#include <conio.h>

void main(void)
{
  char save[]  = "to disk.",
  prompt[]     = " File exist,overwrite?[y/n]",
  err[]      = "Error occured. ",
  newline[]  = "\n\r";

int handle, ch;
unsigned count;

  /* Open file and create and overwrite it */

  if (_dos_createnew( "DUMMY.FIL",_A_NORMAL, &handle) !=0)
  {
    if (errno == EEXIST)
    {
    /* Use _dos_write to display prompts*/
    _dos_write (1, prompt, sizeof( prompt ) -1, &count );
    ch = bdos( 1, 0, 0) & 0x00ff;
    if ( (ch == 'y') || (ch == 'Y') )
      _dos_creat( "DUMMY.FIL", _A_NORMAL, &handle);
    _dos_write( 1,newline, sizeof( newline) -1, &count);
    }
  }

  /* Write to file; output passes through operating system's buffer*/

  if ( _dos_write( handle, save, sizeof( save ),&count) != 0 )

  {
  _dos_write( 1, err, sizeof( err) - 1, &count );
  _dos_write( 1, newline, sizeof( newline ) -1, &count );
  }

  /* Write directly to file with no intermediate buffering */
  if ( _dos_commit( handle ) != 0 )
  {
  _dos_write( 1, err, sizeof(err) -1, &count);
  _dos_write( 1, newline, sizeof( newline ) - 1 , &count);
  }

  /* Close file */
  if (_dos_close( handle ) != 0)
  {
  _dos_write( 1, err, sizeof(err) -1, &count );
  _dos_write( 1, newline, sizeof(newline) -1, &count );
  }

}
```

/* _expand example */

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main(void)
{
  char *bufchar;

  printf( "Allocate a 512 element buffer\n" );
  if( (bufchar = (char *) calloc(512, sizeof( char ) )) == NULL)
    exit( 1 );
  printf( "Allocated %d bytes at %Fp\n",
    _msize ( bufchar ), (void __far *)bufchar );

  if ((bufchar = (char *) _expand (bufchar, 1024)) == NULL)
    printf ("can not expand");
  else
    printf (" Expanded block to %d bytes at %Fp\n",
      _msize( bufchar ) , (void __far *)bufchar );
/* free memory  */
  free( bufchar );
  exit (0);
}
```

/* _get_osfhandle and _open_osfhandle example */

```
#include <windows.h>
#include <fcntl.h>
#include <stdio.h>
#include <io.h>
#ifndef __FLAT__
#error This Example must be compiled using 32 bit compiler
#endif

//Example for _get_osfhandle() and _open_osfhandle()

BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
Example_get_osfhandle(HWND hWnd);

#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)

{
MSG msg;      // message

  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize

  /* Perform initializations that apply to a specific instance */

  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);


  /* Acquire and dispatch messages until a WM_QUIT message is received. */

  while (GetMessage(&msg, // message structure
   NULL, // handle of window receiving the message
   NULL, // lowest message to examine
   NULL))  // highest message to examine
  {
  TranslateMessage(&msg); // Translates virtual key codes
  DispatchMessage(&msg);  // Dispatches message to window
  }

  return (msg.wParam);  // Returns the value from PostQuitMessage


}
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;

  // Fill in window class structure with parameters that describe the
  // main window.

  wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
```

```
  wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int,
  long ))MainWndProc; // Function to retrieve messages for
                // windows of this class.
  wc.cbClsExtra = 0;  // No per-class extra data.
  wc.cbWndExtra = 0;  // No per-window extra data.
  wc.hInstance = hInstance; // Application that owns the class.
  wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
  wc.lpszClassName = "Example"; // Name used in call to CreateWindow.

  /* Register the window class and return success/failure code. */

  return (RegisterClass(&wc));

}
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND  hWnd; // Main window handle.

  /* Create a main window for this application instance.  */

  hWnd = CreateWindow(
   "Example",                  // See RegisterClass() call.
   "Example _get_osfhandle _open_osfhandle (32 bit)",  // Text for window
  title bar.
   WS_OVERLAPPEDWINDOW,        // Window style.
   CW_USEDEFAULT,          // Default horizontal position.
   CW_USEDEFAULT,           // Default vertical position.
   CW_USEDEFAULT,             // Default width.
   CW_USEDEFAULT,         // Default height.
   NULL,                   // Overlapped windows have no parent.
   NULL,                   // Use the window class menu.
   hInstance,               // This instance owns this window.
   NULL                    // Pointer not needed.
  );

  /* If window could not be created, return "failure" */

  if (!hWnd)
   return (FALSE);

  /* Make the window visible; update its client area; and return "success"
  */

  ShowWindow(hWnd, nCmdShow); // Show the window
  UpdateWindow(hWnd);     // Sends WM_PAINT message
  return (hWnd);        // Returns the value from PostQuitMessage

}
#pragma argsused
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{

  switch (message)
```

```
  {
    case WM_CREATE:
    {
      Example_get_osfhandle(hWnd);
      return NULL;
    }
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;

    default:      // Passes it on if unproccessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}

Example_get_osfhandle(HWND hWnd)
{
    long osfHandle;
    char str[128];
    int fHandle = open("file1.c", O_CREAT|O_TEXT);
    if(fHandle != -1)
    {
      osfHandle = _get_osfhandle(fHandle);
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle,
  osfHandle);
      MessageBox(hWnd,str,"_get_osfhandle",MB_OK|MB_ICONINFORMATION);
      close(fHandle);

      fHandle = _open_osfhandle(osfHandle, O_TEXT );
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle,
  osfHandle);
      MessageBox(hWnd,str,"_open_osfhandle",MB_OK|MB_ICONINFORMATION);
         close(fHandle);

    }
    else
      MessageBox(hWnd,"File Open Error","WARNING",MB_OK|MB_ICONSTOP);
return 0;
}
```

/* _heapset example */

```
#include <windowsx.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

#ifndef __FLAT__
#error This Example must be compiled using 32 bit compiler
#endif
BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
void ExampleHeapSet(HWND hWnd);
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)

{
MSG msg;      // message

  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize

  /* Perform initializations that apply to a specific instance */

  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);


  /* Acquire and dispatch messages until a WM_QUIT message is received. */

  while (GetMessage(&msg, // message structure
   NULL, // handle of window receiving the message
   NULL, // lowest message to examine
   NULL))  // highest message to examine
  {
  TranslateMessage(&msg); // Translates virtual key codes
  DispatchMessage(&msg);  // Dispatches message to window
  }

  return (msg.wParam);  // Returns the value from PostQuitMessage


}
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;

  // Fill in window class structure with parameters that describe the
  // main window.

  wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
  wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int,
  long ))MainWndProc; // Function to retrieve messages for
                // windows of this class.
```

```
  wc.cbClsExtra = 0;  // No per-class extra data.
  wc.cbWndExtra = 0;  // No per-window extra data.
  wc.hInstance = hInstance; // Application that owns the class.
  wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
  wc.lpszClassName = "Example"; // Name used in call to CreateWindow.

  /* Register the window class and return success/failure code. */

  return (RegisterClass(&wc));

}
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND  hWnd; // Main window handle.

  /* Create a main window for this application instance.  */

  hWnd = CreateWindow(
   "Example",                 // See RegisterClass() call.
   "Example _heapset 32 bit only",  // Text for window title bar.
   WS_OVERLAPPEDWINDOW,        // Window style.
   CW_USEDEFAULT,          // Default horizontal position.
   CW_USEDEFAULT,           // Default vertical position.
   CW_USEDEFAULT,             // Default width.
   CW_USEDEFAULT,         // Default height.
   NULL,                   // Overlapped windows have no parent.
   NULL,                   // Use the window class menu.
   hInstance,               // This instance owns this window.
   NULL                    // Pointer not needed.
  );

  /* If window could not be created, return "failure" */

  if (!hWnd)
   return (FALSE);

  /* Make the window visible; update its client area; and return "success"
  */

  ShowWindow(hWnd, nCmdShow); // Show the window
  UpdateWindow(hWnd);     // Sends WM_PAINT message
  return (hWnd);        // Returns the value from PostQuitMessage

}

void ExampleHeapSet(HWND hWnd)
{
  int hsts;
  char *buffer;

  if ( (buffer = (char *)malloc( 1 )) == NULL )
    exit(0);
  hsts = _heapset( 'Z' );
```

```
  switch (hsts)
  {
    case _HEAPOK:
      MessageBox(hWnd,"Heap is OK","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPEMPTY:
      MessageBox(hWnd,"Heap is empty","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPBADNODE:
      MessageBox(hWnd,"Bad node in heap","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    default:
      break;

  }

  free (buffer);
}
#pragma argsused
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{

  switch (message) {
    case WM_CREATE:
    {
      //Example _heapset
      ExampleHeapSet(hWnd);
      return NULL;
    }
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;

    default:      // Passes it on if unproccessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}
```

/* _searchstr example */

```
#include <stdio.h>
#include <stdlib.h>

char buf[_MAX_PATH];

int main(void)
{
  /* look for TLINK.EXE */
  _searchstr("TLINK.EXE", "PATH", buf);
  if (buf[0] == '\0')
  printf ("TLINK.EXE not found\n");
  else
  printf ("TLINK.EXE found in %s\n", buf);

  return 0;
}
```

/* _popen and _pclose example */

```
/* this program initiates a child process to run the dir command
   and pipes the directory listing from the child to the parent.
*/

#include <stdio.h>     // popen() pclose() feof() fgets() puts()
#include <string.h>    // strlen()

int main( )
{
  FILE* handle;        // handle to one end of pipe
  char message[256];   // buffer for text passed through pipe
  int status;          // function return value

  // open a pipe to receive text from a process running "DIR"
  handle = _popen("dir /b", "rt");
  if (handle == NULL)
  {
    perror("_popen error");
  }

  // read and display input received from the child process
  while (fgets(message, sizeof(message), handle))
  {
    fprintf(stdout, message);
  }

  // close the pipe and check the return status
    status = _pclose(handle);
  if (status == -1)
  {
    perror("_pclose error");
  }

  return(0);
}
```

/* wctomb example */

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int x;
  wchar_t wc = L'a';
  char *pmbNULL = NULL;
  char *pmb = (char *)malloc(sizeof( char ));

  printf (" Convert a wchar_t array into a multibyte string:\n");
  x = wctomb( pmb, wc);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string: %1s\n\n",pmb);

  printf (" Convert when target is NULL\n");
  x = wctomb( pmbNULL, wc);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte stri ng: %1s\n\n",pmbNULL);

}
```

/* wcstombs example */

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int x;
  char *pbuf = (char*)malloc( MB_CUR_MAX);
  wchar_t *pwcsEOL = L'\0';
  char *pwchi= L"Hi there!";

  printf (" Convert entire wchar string into a multibyte string:\n");
  x = wcstombs( pbuf, pwchi,MB_CUR_MAX);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string character: %1s\n\n",pbuf);

  printf (" Convert when target is NULL\n");
  x = wcstombs( pbuf, pwcsEOL, MB_CUR_MAX);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte string: %1s\n\n",pbuf);

}
```

Using EasyWin

See also

Borland C++ provides EasyWin, a feature that lets you compile standard DOS applications which use traditional TTY style input and output so they can run as true Windows programs. With EasyWin, you do not need to change a DOS program to run it under Windows.

Note: You cannot use EasyWin with the DLL version of the run-time library.

EasyWin includes:

clreol	gotoxy	wherey
clrscr	wherex	

These functions have the same names (and uses) as functions in conio.h header file. Classes in constrea.h provide console I/O functionality for use with C++ streams.

The following routines can be ported to EasyWin programs but are not available in 16-bit Windows programs:

fgetchar	kbhit	puts
getch	perror	scanf
getchar	printf	vprintf
getche	putch	vscanf
gets	putchar	

These functions are provided to simplify porting of existing DOS code into 16-bit Windows applications.

Converting DOS applications to Windows

C Example C++ Example

To convert console-based applications that use standard files or *iostream* functions, check the EasyWin Target Type using TargetExpert in the IDE. Borland C++ will detect that your program does not contain a *WinMain* function (normally required for Windows applications) and link the EasyWin library. When you run your program in the Windows environment, a standard window is created, and your program takes input and produces output for that window as if it were the standard screen.

You can use the EasyWin window any time to request input to or specify output from a TTY device. This means that in addition to stdin and stdout, all stderr, stdaux, and *cerr* devices are all connected to this window.

EasyWin C example

```
#include <stdio.h>

int main()
{
  printf("Hello Windows\n");
  return 0;
}
```

EasyWin C++ example

```
#include <iostream.h>

int main()
{
  cout << "Hello Windows\n";
  return 0;
}
```

Using EasyWin from within a Windows program

See also Example

Borland C++ provides EasyWin so you can quickly and easily convert your DOS applications to 16-bit Windows programs.

You can also use EasyWin from within 16-bit Windows programs. For example, you can add printf functions to your program code to help debug a Windows program.

To use EasyWin from within a Windows program, call _InitEasyWin() before performing any standard input or output.

_InitEasyWin example

```
#include <stdio.h>
#include <windows.h>

#pragma argsused

int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
{
   char *p;

   _InitEasyWin();

   p = "This is an example of how Borland C++"
       " will automatically\nconcatenate"
       " very long strings,\nresulting in nicer"
       " looking programs.";

   printf(p);

   return(0);
}
```

EasyWin features

See also

EasyWin now has support for several new features:

{bullet.bmp} Printing support lets you print the contents of the EasyWin window.

{bullet.bmp} Viewable scrolling buffer stores either 100 or 400 lines of text (depending on the memory model). This buffer automatically scrolls as you move the vertical or horizontal scroll bar thumb tabs.

{bullet.bmp} Redirects output to a file of your choice when the buffer runs out of space.

{bullet.bmp} Full Windows Clipboard support, lets you paste to standard input and copying from the buffer onto the Clipboard, using either the keyboard or the mouse.

EasyWin: Printing

Use the Print command on the system menu to print the contents of an EasyWin window. It activates the standard Print dialog from which you can specify printing options.

By default, EasyWin prints 80 columns and approximately 54 lines on U.S. Letter size (8.5" x 11") paper.

Note: The Print command is grayed if you do not have a default printer installed under Windows. If you have a printer installed but it is not the default, make it the default printer before attempting to print from an EasyWin application.

If you have trouble printing on a dot-matrix printer, add the following global variable to your main source file:

```
  BOOL _UseDefaultPrinterFont;
```

Set this variable to TRUE and EasyWin will print using the default font for your printer instead of the standard EasyWin printer font.

You should declare this variable as external and set it to TRUE within your *main()* function:

```
extern BOOL _UseDefaultPrinterFont;
   .
   .
   .
int main()
{
  _UseDefaultPrinterFont = TRUE;
     .
     .
     .
}
```

Note: This variable is not recommended for use with laser or inkjet printers.

EasyWin: Scrolling Buffer

EasyWin caches your screen output into a buffer of either:

{bullet.bmp} 400 lines (for compact and large memory models)

{bullet.bmp} 100 lines (for small and medium memory models)

You can view the buffer any time by using the scroll bar or any of the standard window movement keys.

You can change the buffer size of your EasyWin application by declaring the following global variable in your main source file with the appropriate initializer:

```
POINT _BufferSize = { X, Y };
```

where:

`X` is the number of columns you want. Setting `X` to a value other than 80 is not recommended as the results are unpredictable.

`Y` is the number of lines you want. If you need to specify a value for `Y` greater than 100, use the compact or large memory model. The small and medium memory models have limited local heap space for the buffer.

Autoscrolling

If you click and drag either the vertical or horizontal scroll bar thumb tab, the text in the buffer automatically scrolls up and down or left and right. This is a useful feature when you want to quickly scan large amounts of data in the EasyWin window.

EasyWin: saving text in an output file

If you want to redirect the output of your program to a file, add the following global variable to your main source file:

```
extern char *_OutputFileName = "C:\\myoutput.txt";
```

Make `_OutputFileName` the name of the file in which to store the redirected output.

Note: If the output file you specified already exists, it is deleted without warning.

EasyWin: Clipboard support

EasyWin lets you to cut, copy, and paste text from an EasyWin application window.

To select text, use the Edit command from the system menu and choose Mark. This puts you in Mark mode. You can use the mouse or the keyboard to select text. You can move the cursor and select text using the standard rules and keystrokes for this feature.

Action	Explanation
Enter	Exits Mark mode. Any marked text is copied to the Clipboard.
Escape	Exits Mark mode. No text is selected.
Right mouse button	same as Enter.
Edit|Copy	same as Enter.
Edit|Paste	pastes text into *stdin*, receiving the contents of the Clipboard as input to your program, merging it with any keyboard input.

Example

If you are writing a program that requests its data from the keyboard via *scanf*, *cin*, or other similar *stdio/conio* functions:

1. Write a data file that contains your entire input.
2. Load that file into NotePad, select it, and copy it to the Clipboard.
3. Run your program, go to the system edit menu, and choose Paste.

Your program accepts the contents of Clipboard as input.

Notes:

{bullet.bmp} The Paste command is grayed if the Clipboard contains no objects of type CF_TEXT or if your program has terminated.

{bullet.bmp} The Copy command is grayed if you have not selected a block of text.

International API overview

See also

The Borland C++ provides support for developing international applications. The Borland C++ runtime library now includes extensions to many of the single-byte routines. These extensions allow you to write applications that can process multibyte or Unicode types.

International API routines

See also

To allow maximum portability, Borland C++ provides a portable macro for that expands to a multibyte or a Unicode routine without having to rewrite the source code. When you use the portable macros, you can recompile and define one of the following macros.

_MBCS	enables multibyte routines
_UNICODE	enables wide-character routines

If neither macro is defined, the single-byte routines are used.

The following table provides a list of the routines that are available for international applications. The column **Unicode platform support** provides a list of the functions that are not supported on Windows NT. Some Unicode functions are available as macros. When a routine is available as a macro, the macro version is used by default. To get the function version of a routine, you must undefine the macro.

Single byte	Portable macro	Multibyte	Unicode	Unicode platform support
	_istlegal	_ismbclegal	-	Win 95, NT
	_istlead	_ismbblead	-	Win 95, NT
	_isleadbyte	_ismbblead	-	Win 95, NT
_argv	_targv		_wargv	Win NT
_atoi64	_ttoi64		_wtoi64	Win 95, NT
_atold	_ttold		_wtold	Win 95, NT
closedir	_tclosedir		wclosedir	WIN NT
_environ	_tenviron		_wenviron	Win NT
_fdopen	_tfdopen		_wfdopen	Win 95, NT
_fsopen	_tfsopen		_wfsopen	Win NT
_fullpath	_tfullpath		_wfullpath	Win NT
_getdcwd	_tgetdcwd		_wgetdcwd	Win NT
_i64toa	_i64tot		_i64tow	Win 95, NT
_makepath	_tmakepath		_wmakepath	Win 95, NT
_popen	_tpopen		_wpopen	Win NT
readdir	_treaddir		wreaddir	WIN NT
_rtl_chmod	_trtl_chmod		_wrtl_chmod	Win NT
_rtl_creat	_trtl_creat		_wrtl_creat	Win NT
_rtl_open	_trtl_open		_wrtl_open	Win NT
_searchenv	_tsearchenv		_wsearchenv	Win NT
_searchstr	_tsearchstr		_wsearchstr	Win NT
_snprintf	_sntprintf		_snwprintf	Win 95, NT
_splitpath	_tsplitpath		_wsplitpath	Win 95, NT

_strdate	_tstrdate		_wstrdate	Win 95, NT
_strdec	_tcsdec	_mbsdec	_wcsdec	Win 95, NT
_stricoll	_tcsicoll	_mbsicoll	_wcsicoll	Win 95, NT
_strinc	_tcsinc	_mbsinc	_wcsinc	Win 95, NT
_strncnt	_tcsnbcnt	_mbsnbcnt	_wcsncnt	Win 95, NT
_strncoll	_tcsnccoll	_mbsncoll	_wcsncoll	Win 95, NT
_strncoll	_tcsncoll	_mbsnbcoll	_wcsncoll	Win 95, NT
_strnextc	_tcsnextc	_mbsnextc	_wcsnextc	Win 95, NT
_strnicoll	_tcsncicoll	_mbsnbicoll	_wcsnicoll	Win 95, NT
_strnicoll	_tcsnicoll	_mbsnbicoll	_wcsnicoll	Win 95, NT
_strninc	_tcsninc	_mbsninc	_wcsninc	Win 95, NT
_strspnp	_tcsspnp	_mbsspnp	_wcsspnp	Win 95, NT
_strtime	_tstrtime		_wstrtime	Win 95, NT
_strtold	_tcstold		_wcstold	Win 95, NT
_tzname	_ttzname		_wtzname	Win NT
_ui64toa	_ui64tot		_ui64tow	Win 95, NT
access	_taccess		_waccess	Win NT
asctime	_tasctime		_wasctime	Win 95, NT
atof	_ttof		_wtof	Win 95, NT
atoi	_ttoi		_wtoi	Win 95, NT
atol	_ttol		_wtol	Win 95, NT
chdir	_tchdir		_wchdir	Win NT
chmod	_tchmod		_wchmod	Win NT
creat	_tcreat		_wcreat	Win NT
ctime	_tctime		_wctime	Win 95, NT
execl	_texecl		_wexecl	Win NT
execle	_texecle		_wexecle	Win NT
execlp	_texeclp		_wexeclp	Win NT
execlpe	_texeclpe		_wexeclpe	Win NT
execv	_texecv		_wexecv	Win NT
execve	_texecve		_wexecve	Win NT
execvp	_texecvp		_wexecvp	Win NT
execvpe	_texecvpe		_wexecvpe	Win NT
fgetc	_fgettc		fgetwc	Win 95, NT
_fgetchar	_fgettchar		_fgetwchar	Win 95, NT

fgets	_fgetts		fgetws	Win 95, NT
findfirst	_tfindfirst		_wfindfirst	Win NT
findnext	_tfindnext		_wfindnext	Win NT
fnmerge	_tfnmerge		_wfnmerge	Win NT
fnsplit	_tfnsplit		_wfnsplit	Win NT
fopen	_tfopen		_wfopen	Win NT
fprintf	_ftprintf		fwprintf	Win 95, NT
fputc	_fputtc		fputwc	Win 95, NT
_fputchar	_fputtchar		_fputwchar	Win 95, NT
fputs	_fputts		fputws	Win 95, NT
freopen	_tfreopen		_wfreopen	Win NT
getc	_gettc		getwc	Win 95, NT
getchar	_gettchar		getwchar	Win 95, NT
getcurdir	_tgetcurdir		_wgetcurdir	Win NT
getcwd	_tgetcwd		_wgetcwd	Win NT
getenv	_tgetenv		_wgetenv	Win 95, NT
gets	_getts		_getws	Win 95, NT
isalnum	_istalnum	_ismbcalnum	iswalnum	Win 95, NT
isalpha	_istalpha	_ismbcalpha	iswalpha	Win 95, NT
isascii	_istascii		iswascii	Win 95, NT
iscntrl	_istntrl		iswcntrl	Win 95, NT
isdigit	_istdigit	_ismbcdigit	iswdigit	Win 95, NT
isgraph	_istgraph	_ismbcgraph	iswgraph	Win 95, NT
islower	_istlower	_ismbclower	iswlower	Win 95, NT
isprint	_istprint	_ismbcprint	iswprint	Win 95, NT
ispunct	_istpunct	_ismbcpunct	iswpunct	Win 95, NT
isspace	_istspace	_ismbcspace	iswspace	Win 95, NT
isupper	_istupper	_ismbcupper	iswupper	Win 95, NT
isxdigit	_istxdigit		iswxdigit	Win 95, NT
ltoa	_ltot		_ltow	Win 95, NT
main	_tmain		wmain	Win NT
memchr	_tmemchr		_wmemchr	Win 95, NT
memcpy	_tmemcpy		_wmemcpy	Win 95, NT
memset	_tmemset		_wmemset	Win 95, NT
mkdir	_tmkdir		_wmkdir	Win NT

_mktemp	_tmktemp		_wmktemp	Win 95, NT
open	_topen		_wopen	Win NT
opendir	_topendir		wopendir	WIN NT
perror	_tperror		_wperror	Win 95, NT
printf	_tprintf		wprintf	Win 95, NT
putc	_puttc		putwc	Win 95, NT
putchar	_puttchar		putwchar	Win 95, NT
putenv	_tputenv		_wputenv	Win NT
puts	_putts		_putws	Win 95, NT
remove	_tremove		wremove	Win NT
rename	_trename		_wrename	Win NT
rewinddir	_trewinddir		wrewinddir	WIN NT
_rmdir	_trmdir		_wrmdir	Win NT
scanf	_tscanf		wscanf	Win 95, NT
searchpath	_tsearchpath		wsearchpath	Win NT
setlocale	_tsetlocale		_wsetlocale	Win 95, NT
_sopen	_tsopen		_wsopen	Win 95, NT
spawnl	_tspawnl		_wspawnl	Win NT
spawnle	_tspawnle		_wspawnle	Win NT
spawnlp	_tspawnlp		_wspawnlp	Win NT
spawnlpe	_tspawnlpe		_wspawnlpe	Win NT
spawnv	_tspawnv		_wspawnv	Win NT
spawnve	_spawnve		_wspawnve	Win NT
spawnvp	_tspawnvp		_wspawnvp	Win NT
spawnvpe	_tspawnvpe		_wspawnvpe	Win NT
sprintf	_stprintf		swprintf	Win 95, NT
sscanf	_stsscanf		swscanf	Win 95, NT
stat	_tstat		_wstat	Win NT
_stpcpy	_tcspcpy		_wcspcpy	Win 95, NT
strcat	_tcscat	_mbscat	wcscat	Win 95, NT
strchr	_tcschr	_mbschr	wcschr	Win 95, NT
strcmp	_tcscmp	_mbscmp	wcscmp	Win 95, NT
strcmpi	_tcscmpi	_mbsicmp	_wcscmpi	Win 95, NT
strcoll	_tcscoll	_mbscoll	wcscoll	Win 95, NT
strcpy	_tcscpy	_mbscpy	wcscpy	Win 95, NT

strcspn	_tcscspn	_mbscspn	wcscspn	Win 95, NT
strdup	_tcsdup	_mbsdup	_wcsdup	Win 95, NT
strftime	_tcsftime		wcsftime	Win 95, NT
_stricmp	_stricmp	_mbsicmp	_wcsicmp	Win 95, NT
strlen	_tcslen	_mbslen	wcslen	Win 95, NT
strlen	_tcsclen	_mbslen	wcslen	Win 95, NT
strlwr	_tcslwr	_mbslwr	_wcslwr	Win 95, NT
strncat	_tcsncat	_mbsnbcat	wcsncat	Win 95, NT
strncat	_tcsnccat	_mbsncat	wcsncat	Win 95, NT
strncmp	_tcsnccmp	_mbsncmp	wcsncmp	Win 95, NT
strncmp	_tcsncmp	_mbsnbcmp	wcsncmp	Win 95, NT
strncmpi	_tcsncmpi		wcsncmpi	Win 95, NT
strncnt	_tcsnccnt	__mbsncnt	_wcsncnt	Win 95, NT
strncnt	_tcsnbcnt	_mbsnbcnt	_wcsncnt	Win 95, NT
strncpy	_tcsncpy	_mbsnbcpy	wcsncpy	Win 95, NT
strncpy	_tcsnccpy	_mbsncpy	wcsncpy	Win 95, NT
strnicmp	_tcsncicmp	_mbsnicmp	_wcsnicmp	Win 95, NT
strnicmp	_tcsnicmp	_mbsnbicmp	wcsnicmp	Win 95, NT
strnset	_tcsnset	_mbsnbset	_wcsnset	Win 95, NT
strnset	_tcsncset	_mbsnset	_wcsnset	Win 95, NT
strpbrk	_tcspbrk	_mbspbr	wcspbrk	Win 95, NT
strrchr	_tcsrchr	_mbsrchr	wcsrchr	Win 95, NT
strrev	_tcsrev	_mbsrev	_wcsrev	Win 95, NT
strset	_tcsset	_mbsset	_wcsset	Win 95, NT
strspn	_tcsspn	_mbsspn	wcsspn	Win 95, NT
strstr	_tcsstr	_mbsstr	wcsstr	Win 95, NT
strtod	_tcstod		wcstod	Win 95, NT
strtok	_tcstok	_mbstok	wcstok	Win 95, NT
strtol	_tcstol		wcstol	Win 95, NT
strtoul	_tcstoul		_wcstoul	Win 95, NT
strupr	_tcsupr	_mbsupr	_wcsupr	Win 95, NT
strxfrm	_tcsxfrm		wcsxfrm	Win 95, NT
system	_tsystem		_wsystem	Win NT
_tempnam	_ttempnam		_wtempnam	Win 95, NT
tmpnam	_ttmpnam		_wtmpnam	Win 95, NT

tolower	_totlower	_mbctolower	towlower	Win 95, NT
toupper	_totupper	_mbctoupper	towupper	Win 95, NT
tzset	_ttzset		_wtzset	Win 95, NT
ultoa	_ultot		_ultow	Win 95, NT
ungetc	_ungettc		ungetwc	Win 95, NT
_unlink	_tunlink		_wunlink	Win NT
_utime	_tutime		_wutime	Win 95, NT
vfprintf	_vftprintf		vfwprintf	Win 95, NT
vprintf	_vtprintf		vwprintf	Win 95, NT
vsprintf	_vstprintf		vswprintf	Win 95, NT
WinMain	_tWinMain		wWinMain	Win NT

Unicode macros

See also

By default, these Unicode routines are available as a macro. To get the function version, you must undefine the macro.

iswalpha

iswascii

iswcntrl

iswdigit

iswgraph

iswlower

iswprint

iswpunct

iswspace

iswupper

iswxdigit

International API formatted I/O

See also

There are now versions of some runtime library functions that take wide strings (**wchar_t***) instead of narrow strings (**char***). These wide functions have similar names as their narrow counter parts but with a *w* placed in it. For example: along with *printf* and *scanf* there are now *wprintf* and *wscanf* functions. The file TCHAR.H has **#define** names that map to either the narrow versions (for normal ANSI **char**'s) or wide versions (for Unicode support) based on the setting of the _UNICODE macro.

The standard functions operate on regular strings, and the wide versions operate on wide strings. The *printf* and *scanf* family of functions allow you to input or output similar width or opposite width strings with some new format conversion characters and prefixes.

The narrow versions of the functions take narrow format strings and default to reading/writing narrow strings and chars. The wide versions of the functions take wide format strings and default to reading/writing wide strings and chars.

Note: The capitol letter version of **%s** and **%c** (**%S** and **%C**) mean "*use the opposite width than the default for the function that was called*". This means that **%S** in *wprintf* will write to a narrow string.

Also, **%l** and **%h** force the width to be either long (wide) or short (narrow).

Summary of formatted I/O functions

See also

Here is a summary of the current *printf* and *scanf* family of functions.

ANSI function	Unicode function	Description
cprintf	{None}	Console output
cscanf	{None}	Console input
fprintf	fwprintf	FILE * stream output
fscanf	fwscanf	FILE * stream input
printf	wprintf	STDOUT output
scanf	wscanf	STDIN input
sprintf	swprintf	string/memory output
sscanf	swscanf	string/memory input
vfprintf	vfwprintf	VA_LIST FILE* stream output
vfscanf	vfwscanf	VA_LIST FILE* stream input
vprintf	vwprintf	VA_LIST STDOUT output
vscanf	vwscanf	VA_LIST STDIN input

Unicode output format specifiers

See also

The following table shows the formatted ouput specifiers for the Unicode family of functions. The table shows how the format specifier is used by *printf* and the Unicode family of output functions to output strings and characters.

Format specifier	***printf* function**	**Unicode function**
%c	narrow	wide
%C	wide	narrow
%hc	narrow	narrow
%hC	narrow	narrow
%lc	wide	wide
%lC	wide	wide
%s	narrow	wide
%S	wide	narrow
%hs	narrow	narrow
%hS	narrow	narrow
%ls	wide	wide
%lS	wide	wide

Unicode family of output functions

The Unicode output family of functions includes the following.

_snprintf

fprintf

sprintf

vfprintf

vprintf

vsprintf

_snprintf

fwprintf

swprintf

vfwprintf

vwprintf

vswprintf

Unicode input format specifiers

See also

The following table shows the formatted ouput specifiers for the Unicode family of functions. The table shows how the format specifier is used by *scanf* and the Unicode family of input functions to input strings and characters.

Format specifier	*scanf* **function**	**Unicode function**
%c	narrow	wide
%C	wide	narrow
%hc	narrow	narrow
%hC	narrow	narrow
%lc	wide	wide
%lC	wide	wide
%s	narrow	wide
%S	wide	narrow
%hs	narrow	narrow
%hS	narrow	narrow
%ls	wide	wide
%lS	wide	wide

Unicode family of input functions

The Unicode input family of functions includes the following.

sscanf

swscanf

Extended types formatted I/O

See also

The following table shows new format specifiers implemented in Borland C++ for the *printf* and *scanf* family of functions. This implementation allows the input and output of 64-bit integers and provides greater I/O flexibility for other types.

Format character	Functionality
%Ld	**__int64**
%I8d	8-bit wide integer (**char**)
%I16d	16-bit wide integer (**short**)
%I32d	32-bit wide integer (**long**)
%I64d	64-bit wide integer (**__int64**)

Note that the above table uses the **%d** format as an example. The **I8**, **I16**, **I32**, **I64** prefixes can be used with the **d**, **i**, **o**, **x**, **X** formats, as well as the new **L** prefix previously allowed only on **float** to specify **long double** type.

BIVBX library functions

This file contains information about using the BIVBX library functions defined in the header file, bivbx.h, located in your include directory. If you are using VBX controls with your C, C++, or ObjectWindows applications, you will need to read this information so that you can use the correct VBX functions to initialize VBX support, return a VBX control handle, initialize a dialog window, handle events, and so forth.

For more information about using VBX controls in your C, C++, or ObjectWindows programs, see the online text file, VBX.TXT, which describes how to use VBXGEN, a utility program designed to generate a header file from a VBX control library.

For more details, see these topics:

- Initialization Functions
- Controls
- Dialogs
- Properties
- Events
- Methods
- Conversions
- Dynamic strings
- Pictures
- Basic strings
- Form files
- 32-bit Issues

Initialization Functions

VBXInit
VBXTerm
VBXEnableDLL

VBXInit

See also

Syntax

```
BOOL VBXInit( HINSTANCE instance, LPCSTR classPrefix )
```

Description

This function initializes VBX support for the program instance <instance> and must be called before any other VBX function. The <classPrefix> argument specifies the string prefix used when registering VBX window classes (NULL defaults to "BiVbx"). This function returns TRUE if successful or FALSE if unable to initialize.

VBXTerm

See also

Syntax

```
void VBXTerm( void )
```

Description

This function terminates VBX support for the current program instance. No other VBX functions should be called after this function.

VBXEnableDLL

See also

Syntax

```
BOOL VBXEnableDLL( HINSTANCE instApp, HINSTANCE instDLL )
```

Description

This function enables VBX support for a DLL. It should be called prior to loading dialog resources from an instance other than the main program. It returns TRUE if successful or FALSE if an error occurs.

Controls

- VBXGetHct1
- VBXGetHwnd
- VBXCreate

VBXGetHct1

See also

Syntax

```
HCTL VBXGetHctl( HWND window )
```

Description

This function returns the VBX control handle associated with the window <window> or NULL if <window> is not a valid VBX control.

VBXGetHwnd

See also

Syntax

```
HWND VBXGetHwnd( HCTL control )
```

Description

This function returns the window handle associated with the VBX control <control> or NULL if <control> is not a valid VBX control.

VBXCreate

See also

Syntax

```
HCTL VBXCreate( HWND windowParent, UINT id,
                LPCSTR library, LPCSTR cls,
                LPCSTR title, DWORD style,
                int x, int y, int w, int h, int file )
```

Description

This function creates a new instance of the control <cls> located in the VBX library <library>. The <style> argument specifies the control window style and can be set to 0 to use the default style. The <file> argument specifies a form file and is should be set to 0 for dynamically created controls. This function returns NULL if it is unable to load the VBX library and create the control. <x>, <y>, <w>, and <h> are related system coordinates.

Dialogs

VBXInitDialogs

VBXInitDialog

See also

Syntax

```
BOOL VBXInitDialog( HWND window, HINSTANCE instance, LPSTR id )
```

Description

This function is used to initialize a dialog window <window> loaded from a resource <id> (located in <instance>) by creating VBX controls for each child window of class VBControl located in the dialog template. It should be called by the dialog procedure when it receives the WM_INITDIALOG message. It returns TRUE if successful, or FALSE if an error occurs. Resource <id> must be of DLGINIT type.

Properties

VBXGetArrayProp
VBXGetArrayPropByName
VBXGetNumProps
VBXGetProp
VBXGetPropByName
VBXGetPropIndex
VBXGetPropName
VBXGetPropNameBuf
VBXGetPropType
VBXIsArrayProp
VBXSetArrayProp
VBXSetArrayPropByName
VBXSetProp
VBXSetPropByName

VBXGetArrayProp

See also

Syntax

```
BOOL VBXGetArrayProp( HCTL control, int index, LPVOID value, int element )
```

Description

This function retrieves the value of element <element> of property <index> of control <control> and places it into the buffer located at <value>. It returns TRUE if successful, or FALSE if an error occurs.

VBXGetArrayPropByName

See also

Syntax

```
BOOL VBXGetArrayPropByName( HCTL control, LPSTR name, LPVOID value, int
  element )
```

Description

This function retrieves the value of element <element> of property <name> of control <control> and places it into the buffer located at <value>. It returns TRUE if successful, or FALSE if an error occurs.

VBXGetNumProps

See also

Syntax

```
int VBXGetNumProps( HCTL control )
```

Description

This function returns the number of properties supported by the control <control> or -1 if an error occurs.

VBXGetProp

See also

Syntax

```
BOOL VBXGetProp( HCTL control, int index, LPVOID value )
```

Description

This function retrieves the value of property <index> of control <control> and places it into the buffer located at <value>. It returns TRUE if successful, or FALSE if an error occurs.

VBXGetPropByName

See also

Syntax

```
ERR VBXGetPropByName( HCTL control, LPSTR name, LPVOID value )
```

Description

This function retrieves the value of property <name> of control <control> and places it into the buffer located at <value>. It returns TRUE if successful, or FALSE if an error occurs.

For both VBXGetProp and VBXGetPropByName, <value> should be large enough to contain the property data type. Enumerated properties must have sizeof(value) >= sizeof(short).

VBXGetPropIndex

See also

Syntax

```
int VBXGetPropIndex( HCTL control, LPCSTR name )
```

Description

This function returns the index of the property <name> of control <control> or -1 if an error occurs.

VBXGetPropName

See also

Syntax

```
LPCSTR VBXGetPropName( HCTL control, int index ) [OBSOLETE: use
  VBXGetPropNameBuf]
```

Description

This function returns the name of property <index> of control <control> or NULL if an error occurs.

VBXGetPropNameBuf

See also

Syntax

```
int VBXGetPropNameBuf( HCTL control, int index, LPSTR buffer, int len )
```

Description

This function copies up to <len> bytes of the name of property <index> of control <control> into <buffer>. It returns the number of bytes copied or 0 if an error occurs.

VBXGetPropType

See also

Syntax

```
USHORT VBXGetPropType( HCTL control, int index )
```

Description

This function returns the type (e.g. PTYPE_BOOL) of property <index> of control <control> or -1 if an error occurs. The property types are:

Type Name	C Type
PTYPE_CSTRING	HSZ
PTYPE_SHORT	short
PTYPE_LONG	long
PTYPE_BOOL	short
PTYPE_COLOR	COLORREF
PTYPE_ENUM	short
PTYPE_REAL	float
PTYPE_XPOS	long (twips)
PTYPE_XSIZE	long (twips)
PTYPE_YPOS	long (twips)
PTYPE_YSIZE	long (twips)
PTYPE_PICTURE	HPIC
PTYPE_BSTRING	HLSTR

VBXIsArrayProp

See also

Syntax

```
BOOL VBXIsArrayProp( HCTL control, int index )
```

Description

This function returns TRUE if the property <index> of control <control> is an array.

VBXSetArrayProp

See also

Syntax

```
BOOL VBXSetArrayProp( HCTL control, int index, LONG value, int element )
```

Description

This function sets the value of element <element> of property <index> of control <control> to <value>.
It returns TRUE if successful, or FALSE if an error occurs.

VBXSetArrayPropByName

See also

Syntax

```
BOOL VBXSetArrayPropByName( HCTL control, LPSTR name, LONG value, int
  element )
```

Description

This function sets the value of element <element> of property <name> of control <control> to <value>. It returns TRUE if successful, or FALSE if an error occurs.

VBXSetProp

See also

Syntax

```
BOOL VBXSetProp( HCTL control, int index, LONG value )
```

Description

This function sets the value of property <index> of control <control> to <value>. It returns TRUE if successful, or FALSE if an error occurs.

VBXSetPropByName

See also

Syntax

```
BOOL VBXSetPropByName( HCTL control, LPSTR name, LONG value );
```

Description

This function sets the value of property <name> of control <control> to <value>. It returns TRUE if successful, or FALSE if an error occurs.

Events

See also

When a VBX control generates an event, it sends a WM_VBXFIREEVENT message to its parent. The <lParam> argument of the message contains a far pointer to a VBXEVENT structure which describes the event:

```
typedef struct VBXEVENT
{
  HCTL      Control;
  HWND      Window;
  int             ID;
  int             EventIndex;
  LPCSTR    EventName;
  int             NumParams;
  LPVOID    ParamList;
} VBXEVENT, FAR * LPVBXEVENT, NEAR * NPVBXEVENT;
```

Control
the handle for the control that caused the event.

Window
the window handle for the above control.

ID
the control identifier for the above window.

EventIndex
the index into the event list for that control.

EventName
the name of the event (click, mouse move, etc.>.

NumParams
the number of arguments passed to the event.

ParamList
a pointer to an array (of size <NumParams>) of event arguments in reverse order (i.e. Arg0 == e->ParamList[e->NumParams-1]).

VBX_EVENTARGNUM

See also

Syntax

```
<type> VBX_EVENTARGNUM(event,type,index)
```

Description

This macro retrieves an argument <index> of type <type> from VBX event <event>. Note that 0 is the first argument index.

Example

```
int x = VBX_EVENTARGNUM(event,int,0);
int y = VBX_EVENTARGNUM(event,int,1);
```

VBX_EVENTARGSTR

See also

Syntax

```
HLSTR VBX_EVENTARGSTR(event,index)
```

Description

This macro retrieves a string argument <index> of type HLSTR from VBX event <event>. Note that 0 is the first argument index. For example:

```
HLSTR s = VBX_EVENTARGSTR(event,2);
```

VBXGetEventIndex

See also

Syntax

```
int VBXGetEventIndex( HCTL control, LPCSTR name )
```

Description

This function returns the index of event <name> of control <index> or -1 if an error occurs.

VBXGetEventName

See also

Syntax

```
LPCSTR VBXGetEventName( HCTL control, int index ) [OBSOLETE: use
  VBXGetEventNameBuf]
```

Description

This function returns the index of event <index> of control <control> or NULL if an error occurs.

VBXGetEventNameBuf

See also

Syntax

```
int VBXGetEventNameBuf( HCTL control, int index, LPSTR buffer, int len )
```

Description

This function copies up to <len> bytes of the name of event <index> of control <control> into <buffer>. It returns the number of bytes copied or 0 if an error occurs.

VBXGetNumEvents

See also

Syntax

```
int VBXGetNumEvents( HCTL control )
```

Description

This function returns the number of events supported by the control <control> or -1 if an error occurs.

Methods

- VBXMethod
- VBXMethodAddItem
- VBXMethodDrag
- VBXMethodMove
- VBXMethodRefresh
- VBXMethodRemoveItem

VBXMethod

See also

Syntax

```
BOOL VBXMethod( HCTL control, int method, long far * args )
```

Description

This function invokes method <method> on control <control> with arguments <args>. Note that this function is not normally called by application programs and is described here as a means of invoking custom control methods. It returns TRUE if successful or FALSE if an error occurs.

VBXMethodAddItem

See also

Syntax

```
BOOL VBXMethodAddItem( HCTL control, int index, LPCSTR item )
```

Description

This function invokes the standard "add item" method on control <control> where <index> is the index of the item to be added (<item>). The exact meaning of "add item" is dependent on the type of VBX control. It returns TRUE if successful or FALSE if an error occurs.

VBXMethodDrag

See also

Syntax

```
BOOL VBXMethodDrag( HCTL control, int action )
```

Description

This function invokes the standard "drag" method on control <control> where <action> is one of the following:

0 - cancel a drag operation

1 - begin a drag operation

2 - "drop" the control at the current location

It returns TRUE if successful or FALSE if an error occurs.

VBXMethodMove

See also

Syntax

```
BOOL VBXMethodMove( HCTL control, long x, long y, long w, long h )
```

Description

This function invokes the standard "move" method on control <control>. The default behaviour for this method is to position the control at <x>, <y>, <w>, and <h>. It returns TRUE if successful or FALSE if an error occurs.

VBXMethodRefresh

See also

Syntax

```
BOOL VBXMethodRefresh( HCTL control )
```

Description

This function invokes the standard "refresh" method on control <control>. The default behaviour for this method is to update the contents of the control window before returning. It returns TRUE if successful or FALSE if an error occurs.

VBXMethodRemoveItem

See also

Syntax

```
BOOL VBXMethodRemoveItem( HCTL control, int item )
```

Description

This function invokes the standard "remove item" method on control <control> where <index> is the index of the item to be removed. The exact meaning of "remove item" is dependent on the type of VBX control. It returns TRUE if successful or FALSE if an error occurs.

Conversions

See also

VBX controls make use of a combination of Twips and pixel measurements. The following functions are used to convert between these different measurement units:

- VBXTwp2PixY
- VBXTwp2PixX
- VBXPix2TwpY
- VBXPix2TwpX

VBXTwp2PixY

See also

Syntax

```
SHORT VBXTwp2PixY( LONG twips )
```

Description

This function converts a Y coordinate <twips> from twips to pixels.

VBXTwp2PixX

See also

Syntax

```
SHORT VBXTwp2PixX( LONG twips )
```

Description

This function converts an X coordinate <twips> from twips to pixels.

VBXPix2TwpY

See also

Syntax

```
LONG VBXPix2TwpY( SHORT pixels )
```

Description

This function converts a Y coordinate <pixels> from pixels to twips.

VBXPix2TwpX

See also

Syntax

```
LONG VBXPix2TwpX( SHORT pixels )
```

Description

This function converts an X coordinate <pixels> from pixels to twips.

Dynamic strings

See also

VBX controls make extensive use of moveable zero-terminated strings, or "dynamic strings". The following functions are used to manipulate those strings:

- VBXCreateCString
- VBXGetCStringLength
- VBXGetCStringPtr
- VBXGetCStringBuf
- VBXDestroyCString
- VBXLockCString
- VBXLockCStringBuf
- VBXUnlockCString

VBXCreateCString

See also

Syntax

```
HSZ VBXCreateCString( HANDLE segment, LPSTR string )
```

Description

This function creates a new string by allocating from the local heap in and initializing to <string>. It returns 0 if an error occurs.

VBXGetCStringLength

See also

Syntax

```
int VBXGetCStringLength( HSZ string )
```

Description

This function returns the length of dynamic string <string> or 0 if an error occurs.

VBXGetCStringPtr

See also

Syntax

```
LPSTR VBXGetCStringPtr( HSZ string ) [OBSOLETE: use VBXGetCStringBuf]
```

Description

This function returns a pointer to the contents of the dynamic string <string> or 0 if an error occurs.

VBXGetCStringBuf

See also

Syntax

int VBXGetCStringBuf(HSZ string, LPSTR buffer, int len)

Description

This function copies up to <len> bytes of the dynamic string <string> into <buffer>. It returns the number of bytes copied or 0 if an error occurs.

VBXDestroyCString

See also

Syntax

HSZ VBXDestroyCString(HSZ string)

Description

This function destroys the dynamic string <string>. It returns <string>.

VBXLockCString

See also

Syntax

LPSTR VBXLockCString(HSZ string) [OBSOLETE: use VBXLockCStringBuf]

Description

This function locks the string <string> and returns a pointer to the contents or 0 if an error occurs.

VBXLockCStringBuf

See also

Syntax

int VBXLockCStringBuf(HSZ string, LPSTR buffer, int len)

Description

This function locks the dynamic string <string> and copies up to <len> bytes of the string contents into <buffer>. It returns the number of bytes copied or 0 if an error occurs.

VBXUnlockCString

See also

Syntax

void VBXUnlockCString(HSZ string)

Description

This function unlocks the string <string>.

Pictures

See also

VBX controls can support a variety of "picture" property types, including bitmaps, metafiles, and icons. These types are represented by a single structure which contains a union of the different types:

```
typedef struct PICTURE
{
  BYTE Type;
  union
  {
    struct
    {
      HBITMAP  Bitmap;
      HPALETTE Palette;
    } Bitmap;
    struct
    {
      HANDLE Metafile;
      int xExtent;
      int yExtent;
    } Metafile;
    struct
    {
      HICON Icon;
    } Icon;
  } Data;
  BYTE Unused0;
  BYTE Unused1;
  BYTE Unused2;
  BYTE Unused3;
} PICTURE, FAR * LPPICTURE, NEAR * NPPICTURE;

#define PICTURE_EMPTY     0
#define PICTURE_BMP         1
#define PICTURE_META       2
#define PICTURE_ICON        3
```

VBXCreatePicture

See also

Syntax

HPIC VBXCreatePicture(LPPICTURE picture)

Description

This function creates and returns a new picture handle from a picture buffer <picture> or 0 if an error occurs.

Example

The following code creates an icon picture:

```
PICTURE pic;
HPIC hpic;
pic.Type = PICTURE_ICON;
pic.Icon.Icon.Icon = LoadIcon( NULL, IDI_ASTERISK );
hpic = VBXCreatePicture( &pic );
```

VBXDestroyPicture

See also

Syntax

void VBXDestroyPicture(HPIC pic)

Description

This function decrements the reference count on the picture handle <pic> and destroys it if the count becomes 0.

VBXGetPicture

See also

Syntax

HPIC VBXGetPicture(HPIC pic, LPPICTURE picture)

Description

This function copies the contents of the picture handle <pic> into the picture buffer <picture> and returns <pic> if successful or 0 if an error occurs.

VBXGetPictureFromClipboard

See also

Syntax

ERR VBXGetPictureFromClipboard(HPIC FAR *pic, HANDLE data, WORD format)

Description

This function creates a new picture handle <*pic> from a clipboard data handle <data> and format <format> and returns non-zero if an error occurs. Valid clipboard formats include CF_BITMAP, CF_METAFILEPICT, CF_DIB and CF_PALETTE.

VBXReferencePicture

See also

Syntax

HPIC VBXReferencePicture(HPIC pic)

Description

This function increments the reference count on the picture handle <pic> and returns <pic> if successful or 0 if an error occurs.

Basic strings

See also

VBX controls make use of moveable string buffers (not zero terminated), or "Basic strings." The following functions are used to manipulate those strings:

- VBXCreateBasicString
- VBXGetBasicStringPtr
- VBXGetBasicStringBuf
- VBXDestroyBasicString
- VBXGetBasicStringLength
- VBXSetBasicString

VBXCreateBasicString

See also

Syntax

HLSTR VBXCreateBasicString(LPVOID buffer, USHORT len)

Description

This function creates a Basic string of length <len> and initial contents of <buffer>. It returns 0 if an error occurs.

VBXGetBasicStringPtr

See also

Syntax

LPSTR VBXGetBasicStringPtr(HLSTR string) [OBSOLETE: use VBXGetBasicStringBuf]

Description

This function returns a pointer to the contents of the Basic string <string> or 0 if an error occurs.

VBXGetBasicStringBuf

See also

Syntax

int VBXGetBasicStringBuf(HLSTR string, LPSTR buffer, int len)

Description

This function copies up to <len> bytes of the Basic string <string> into <buffer>. It returns the number of bytes copied or 0 if an error occurs.

VBXDestroyBasicString

See also

Syntax

void VBXDestroyBasicString(HLSTR string)

Description

This function destroys the Basic string <string>.

VBXGetBasicStringLength

See also

Syntax

USHORT VBXGetBasicStringLength(HLSTR string)

Description

This function returns the length of the Basic string <string> or 0 if an error occurs.

VBXSetBasicString

See also

Syntax

ERR VBXSetBasicString(HLSTR far * string, LPVOID buffer, USHORT len)

Description

This function replaces the contents of the Basic string <string> with <len> bytes from <buffer>. It returns non-zero if an error occurs.

Form Files

See also

These functions are for use with the header files generated by VBXGEN:

VBXCreateFormFile

VBXDeleteFormFile

VBXCreateFormFile

See also

Syntax

HFORMFILE VBXCreateFormFile(LONG len, LPVOID data)

Description

This function creates a temporary form file from a buffer <data> of <len> bytes of data. The form file returned can be used as an argument to the VBXCreate() function. It returns -1 if an error occurs.

VBXDeleteFormFile

See also

Syntax

BOOL VBXDeleteFormFile(HFORMFILE file)

Description

This function deletes the form file <file> and frees any resources associate with it. It returns TRUE if successful, or FALSE if an error occurs.

32-bit Issues

TBvxEventHandler as a base class
Choosing data types
Windows NT

TVbxEventHandler as a base class

See also

ObjectWindows windows and dialogs, including those generated by AppExpert, which use VBX controls must have TVbxEventHandler as a base class if built as a 32-bit application. This can done manually or with ClassExpert.

To manually make a TVbxEventHandler as a base class:

```
class TMyDialog : public TDialog
  .
  .
  .
DEFINE_RESPONSE_TABLE1(TMyDialog, TDialog)
```

should be

```
class TMyDialog : public TDialog, public TVbxEventHandler
  .
  .
  .
DEFINE_RESPONSE_TABLE2(TMyDialog, TDialog, TVbxEventHandler)
```

To make a TVbxEventHandler as a base class using ClassExpert

1. Select the target in the project window
2. Select View | Class Expert
3. Select the desired window or dialog class in the Classes window
4. Select the Control Notifications item in the Events window
5. Select a VBX control under the Control Notifications item
6. Select a VBX event under the VBX control item
7. Use a local menu to add a handler for the selected event
8. ClassExpert will make sure that the window or dialog class is derived from TVbxEventHandler

If TVbxEventHandler is not used as a base class, the VBX control will not appear.

Choosing data types

See also

It's important to use the correct data type when getting property values from a VBX control. This is not always obvious. For example, the following code looks quite reasonable and works in 16-bit:

```
int count;
VBXGetPropByName( hCtl, "Count", &count );
```

This same code will not work in 32-bit since <count> is now 32-bits wide and the emulator (which is 16-bit) only writes 16-bits of information. The lower 2 bytes of <count> are left uninitialized. The best way to fix this is to make <count> a short:

```
short count;
VBXGetPropByName( hCtl, "Count", &count );
```

Please refer to the table in Section 4fix for appropriate data types.

Windows NT

See also

VBX events are not forwarded to ObjectWindows child objects under Windows NT. They are, however, correctly forwarded to the parent object.