beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 9, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 beginner.guide

Copyright (c) 1994-1995, Jason R. Hulance.

A Beginner's Guide to Amiga E
****

This Guide gives an introduction to the Amiga E programming language and, to some extent, programming in general.

Part One: Getting Started

Part Two: The E Language

1.2 beginner.guide/Introduction to Amiga E

Introduction to Amiga E
****

To interact with your Amiga you need to speak a language it understands. Luckily, there is a wide choice of such languages, each of which fits a particular need. For instance, BASIC (in most of its flavours) is simple and easy to learn, and so is ideal for beginners. Assembly, on the other hand, requires a lot of effort and is quite tedious, but can produce the fastest programs so is generally used by commercial programmers. These are two extremes and most businesses and colleges use C or Pascal/Modula-2, which try to strike a balance between simplicity and speed.

E programs look very much like Pascal or Modula-2 programs, but E is based more closely on C. Anyone familiar with these languages will easily learn E, only really needing to get to grips with E's unique features and those borrowed from other languages. This guide is aimed at people who haven't done much programming and may be too trivial for competent programmers, who should find the 'E Reference Manual' more than adequate (although some of the later sections offer different explanations to the Reference Manual, which may prove useful).

Part One (this part) goes through some of the basics of the E language and programming in general. Part Two delves deeper into E, covering the more complex topics and the unique features of E. Part Three goes through a few example programs, which are a bit longer than the examples in the other Parts. Finally, Part Four contains the Appendices, which is where you'll find some other, miscellaneous information.

A Simple Program

1.3 beginner.guide/A Simple Program

A Simple Program
================

If you're still reading you're probably desperate to do some programming in E but you don't know how to start. We'll therefore jump straight in the deep end with a small example. You'll need to know two things before we start: how to use a text editor and the Shell/CLI.

The code

Compilation

Execution

1.4 beginner.guide/The code

The code

Enter the following lines of code into a text editor and save it as the file simple.e (taking care to copy each line accurately). (Just type the characters shown, and at the end of each line press the RETURN or ENTER key.)

```
PROC main()
  WriteF('My first program')
ENDPROC
```

Don't try to do anything different, yet, to the code: the case of the letters in each word is significant and the funny characters are important. If you're a real beginner you might have difficulty finding the ' character. On my GB keyboard it's on the big key in the top left-hand corner directly below the ESC key. On a US and most European keyboards it's two to the right of the L key, next to the ; key.

1.5 beginner.guide/Compilation

Compilation

Once the file is saved (preferably in the RAM disk, since it's only a small program), you can use the E compiler to turn it into an executable program. All you need is the file ec in your C: directory or somewhere else on your search path (advanced users note: we don't need the Emodules: assignment because we aren't using any modules). Assuming you have this and you have a Shell/CLI running, enter the following at the prompt after changing directory to where you saved your new file:

```
ec simple
```

If all's well you should be greeted, briefly, by the E compiler. If anything went wrong then double-check the contents of the file simple.e, that your CLI is in the same directory as this file, and that the program ec is in your C: directory (or on your search path).

1.6 beginner.guide/Execution

Execution

Once everything is working you can run your first program by entering the following at the CLI prompt:

```
simple
```

As a help here's the complete transcript of the whole compilation and execution process (the CLI prompt, below, is the bit of text beginning with 1. and ending in >):

```
1.System3.0:> cd ram:
1.Ram Disk:> ec simple
Amiga E Compiler/Assembler/Linker/PP v3.1a registered (c) '91-95 Wouter
lexical analysing ...
parsing and compiling ...
no errors
1.Ram Disk:> simple
My first program1.Ram Disk:>
```

Your display should be something similar if it's all worked. Notice how the output from the program runs into the prompt (the last line). We'll fix this soon.

1.7 beginner.guide/Understanding a Simple Program

Understanding a Simple Program
****

To understand the example program we need to understand quite a few things. The observant amongst you will have noticed that all it does is print out a message, and that message was part of a line we wrote in the program. The first thing to do is see how to change this message.

Changing the Message

Procedures

Parameters

Strings

Style Reuse and Readability

The Simple Program

1.8 beginner.guide/Changing the Message

Changing the Message
====================

Edit the file so that line contains a different message between the two ' characters and compile it again using the same procedure as before. Don't use any ' characters except those around the message. If all went well, when you run the program again it should produce a different message. If something went wrong, compare the contents of your file with the original and make sure the only difference is the message between the ' characters.

Tinkering with the example

Brief overview

1.9 beginner.guide/Tinkering with the example

Tinkering with the example

Simple tinkering is a good way to learn for yourself so it is encouraged on these simple examples. Don’t stray too far, though, and if you start getting confused return to the proper example pretty sharpish!

1.10 beginner.guide/Brief overview

Brief overview

We’ll look in detail at the important parts of the program in the following sections, but we need first to get a glimpse of the whole picture. Here’s a brief description of some fundamental concepts:

* Procedures: We defined a procedure called main and used the (built-in) procedure WriteF. A procedure can be thought of as a small program with a name.

* Parameters: The message in parentheses after WriteF in our program is the parameter to WriteF. This is the data which the procedure should use.

* Strings: The message we passed to WriteF was a series of characters enclosed in ’ characters. This is known as a string.

1.11 beginner.guide/Procedures

Procedures
==========

As mentioned above, a procedure can be thought of as a small program with a name. In fact, when an E program is run the procedure called main is executed. Therefore, if your E program is going to do anything you must define a main procedure. Other (built-in or user-defined) procedures may be run (or called) from this procedure (as we did WriteF in the example). For instance, if the procedure fred calls the procedure barney the code (or mini-program) associated with barney is executed. This may involve calls to other procedures, and when the execution of this code is complete the next piece of code in the procedure fred is executed (and this is generally the next line of the procedure). When the end of the procedure main has been reached the program has finished. However, lots can happen between the beginning and end of a procedure, and sometimes the program may never get to finish. Alternatively, the program may crash, causing strange things to happen to your computer.

Procedure Definition

Procedure Execution

Extending the example

1.12 beginner.guide/Procedure Definition

Procedure Definition

Procedures are defined using the keyword PROC, followed by the new procedure's name (in lowercase letters), a description of the parameters it takes (in parentheses), a series of lines forming the code of the procedure and then the keyword ENDPROC. Look at the example program again to identify the various parts. See
The code
.

1.13 beginner.guide/Procedure Execution

Procedure Execution

Procedures can be called (or executed) from within the code part of another procedure. You do this by giving its name, followed by some data in parentheses. Look at the call to WriteF in the example program. See

The code
.

1.14 beginner.guide/Extending the example

Extending the example

Here's how we could change the example program to define another procedure:

```
PROC main()
  WriteF('My first program')
  fred()
ENDPROC

PROC fred()
```

```
    WriteF('...slightly improved')
  ENDPROC
```

This may seem complicated, but in fact it's very simple. All we've done is define a second procedure called fred which is just like the original program--it outputs a message. We've called this procedure in the main procedure just after the line which outputs the original message. Therefore, the message in fred is output after this message. Compile the program as before and run it so you don't have to take my word for it.

1.15 beginner.guide/Parameters

Parameters

Generally we want procedures to work with particular data. In our example we wanted the WriteF procedure to work on a particular message. We passed the message as a parameter (or argument) to WriteF by putting it between the parentheses (the (and) characters) that follow the procedure name. When we called the fred procedure, however, we did not require it to use any data so the parentheses were left empty.

When defining a procedure when define how much and what type of data we want it to work on, and when calling a procedure we give the specific data it should use. Notice that the procedure fred (like the procedure main) has empty parentheses in its definition. This means that the procedure cannot be given any data as parameters when it is called. Before we can define our own procedure that takes parameters we must learn about variables. We'll do this in the next chapter. See

Global and local variables
.

1.16 beginner.guide/Strings

Strings

A series of characters between two ' characters is known as a string. Almost any character can be used in a string, although the \ and ' characters have a special meaning. For instance, a linefeed is denoted by the two characters \n. We now know how to stop the message running into the prompt. Change the program to be:

```
  PROC main()
    WriteF('My first program\n')
    fred()
  ENDPROC
```

```
     PROC fred()
       WriteF('...slightly improved\n')
     ENDPROC
```

Compile it as before, and run it. You should notice that the messages now appear on lines by themselves, and the second message is separated from the prompt which follows it. We have therefore cured the linefeed problem we spotted earlier (see Execution).

1.17 beginner.guide/Style Reuse and Readability

Style, Reuse and Readability
============================

The example has grown into two procedures, one called main and one called fred. However, we could get by with only one procedure:

```
     PROC main()
       WriteF('My first program\n')
       WriteF('...slightly improved\n')
     ENDPROC
```

What we've done is replace the call to the procedure fred with the code it represents (this is called inlining the procedure). In fact, almost all programs can be easily re-written to eliminate all but the main procedure. However, splitting a program up using procedures normally results in more readable code. It is also helpful to name your procedures so that their function is apparent, so our procedure fred should probably have been named message or something similar. A well-written program in this style can read just like English (or any other spoken language).

Another reason for having procedures is to reuse code, rather than having to write it out every time you use it. Imagine you wanted to print the same, long message fairly often in your program--you'd either have to write it all out every time, or you could write it once in a procedure and call this procedure when you wanted the message printed. Using a procedure also has the benefit of having only one copy of the message to change, should it ever need changing.

1.18 beginner.guide/The Simple Program

The Simple Program
==================

The simple program should now (hopefully) seem simple. The only bit that hasn't been explained is the built-in procedure WriteF. E has many built-in procedures and later we'll meet some of them in detail. The first thing we need to do, though, is manipulate data. This is really what a computer does all the time--it accepts data from some source (possibly the user), manipulates it in some way (possibly storing it somewhere, too) and outputs new data (usually to a screen or printer). The simple example program did all this, except the first two stages were rather trivial. You told the computer to execute the compiled program (this was some user input) and the real data (the message to be printed) was retrieved from the program. This data was manipulated by passing it as a parameter to WriteF, which then did some clever stuff to print it on the screen. To do our own manipulation of data we need to learn about variables and expressions.

1.19 beginner.guide/Variables and Expressions

Variables and Expressions
****

Anybody who's done any school algebra will probably know what a variable is--it's just a named piece of data. In algebra the data is usually a number, but in E it can be all sorts of things (e.g., a string). The manipulation of data like the addition of two numbers is known as an expression. The result of an expression can be used to build bigger expressions. For instance, 1+2 is an expression, and so is 6-(1+2). The good thing is you can use variables in place of data in expressions, so if x represents the number 1 and y represents 5, then the expression y-x represents the number 4. In the next two sections we'll look at what kind of variables you can define and what the different sorts of expressions are.

Variables

Expressions

1.20 beginner.guide/Variables

Variables
=========

Variables in E can hold many different kinds of data (called types). However, before a variable can be used it must be defined, and this is known as declaring the variable. A variable declaration also decides whether the variable is available for the whole program or just during the code of a procedure (i.e., whether the variable is global or local). Finally, the data stored in a variable can be changed using assignments. The following sections discuss these topics in slightly more detail.

1.21 beginner.guide/Variable types

Variable types

In E a variable is a storage place for data (and this storage is part of the Amiga’s RAM). Different kinds of data may require different amounts of storage. However, data can be grouped together in types, and two pieces of data from the same type require the same amount of storage. Every variable has an associated type and this dictates the maximum amount of storage it uses. Most commonly, variables in E store data from the type LONG. This type contains the integers from -2,147,483,648 to 2,147,483,647, so is normally more than sufficient. There are other types, such as INT and LIST, and more complex things to do with types, but for now knowing about LONG is enough.

1.22 beginner.guide/Variable declaration

Variable declaration

Variables must be declared before they can be used. They are declared using the DEF keyword followed by a (comma-separated) list of the names of the variables to be declared. These variables will all have type LONG (later we will see how to declare variables with other types). Some examples will hopefully make things clearer:

```
     DEF x

     DEF a, b, c
```

The first line declares the single variable x, whilst the second declares the variables a, b and c all in one go.

1.23 beginner.guide/Assignment

```
Assignment
----------
```

The data stored by variables can be changed and this is normally done using assignments. An assignment is formed using the variable’s name and an expression denoting the new data it is to store. The symbol := separates the variable from the expression. For example, the following code stores the number two in the variable x. The left-hand side of the := is the name of the variable to be affected (x in this case) and the right-hand side is an expression denoting the new value (simply the number two in this case).

```
     x := 2
```

The following, more complex example uses the value stored in the variable before the assignment as part of the expression for the new data. The value of the expression on the right-hand side of the := is the value stored in the variable x plus one. This value is then stored in x, over-writing the previous data. (So, the overall effect is that x is incremented.)

```
     x := x + 1
```

This may be clearer in the next example which does not change the data stored in x. In fact, this piece of code is just a waste of CPU time, since all it does is look up the value stored in x and store it back there!

```
     x := x
```

1.24 beginner.guide/Global and local variables

```
Global and local variables (and procedure parameters)
-----------------------------------------------------
```

There are two kinds of variable: global and local. Data stored by global variables can be read and changed by all procedures, but data stored by local variables can only be accessed by the procedure to which they are local. Global variables must be declared before the first procedure definition. Local variables are declared within the procedure to which they are local (i.e., between the PROC and ENDPROC). For example, the following code declares a global variable w and local variables x and y.

```
     DEF w

     PROC main()
       DEF x
       x:=2
       w:=1
       fred()
     ENDPROC

     PROC fred()
```

```
  DEF y
  y:=3
  w:=2
ENDPROC
```

The variable x is local to the procedure main, and y is local to fred. The procedures main and fred can read and alter the value of the global variable w, but fred cannot read or alter the value of x (since that variable is local to main). Similarly, main cannot read or alter y.

The local variables of one procedure are, therefore, completely different to the local variables of another procedure. For this reason they can share the same names without confusion. So, in the above example, the local variable y in fred could have been called x and the program would have done exactly the same thing.

```
DEF w

PROC main()
  DEF x
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=3
  w:=2
ENDPROC
```

This works because the x in the assignment in fred can refer only to the local variable x of fred (the x in main is local to main so cannot be accessed from fred).

If a local variable for a procedure has the same name as a global variable then in the rest of the procedure the name refers only to the local variable. Therefore, the global variable cannot be accessed in the procedure, and this is called descoping the global variable.

The parameters of a procedure are local variables for that procedure. We’ve seen how to pass values as parameters when a procedure is called (the use of WriteF in the example), but until now we haven’t been able to define a procedure which takes parameters. Now we know a bit about variables we can have a go:

```
DEF y

PROC onemore(x)
  y:=x+1
ENDPROC
```

This isn’t a complete program so don’t try to compile it. Basically, we’ve declared a variable y (which will be of type LONG) and a procedure onemore. The procedure is defined with a parameter x, and this is just like a (local) variable declaration. When onemore is called a parameter must be supplied, and this value is stored in the (local) variable x before execution of onemore’s code. The code stores the value of x plus one in the (global) variable y. The following are some examples of calling onemore:

```
       onemore(120)
       onemore(52+34)
       onemore(y)
```

A procedure can be defined to take any number of parameters. Below, the procedure addthem is defined to take two parameters, a and b, so it must therefore be called with two parameters. Notice that values stored by the parameter variables (a and b) can be changed within the code of the procedure.

```
     DEF y

     PROC addthem(a, b)
       a:=a+2
       y:=a*b
     ENDPROC
```

The following are some examples of calling addthem:

```
       addthem(120,-20)
       addthem(52,34)
       addthem(y,y)
```

1.25 beginner.guide/Changing the example

Changing the example

Before we change the example we must learn something about WriteF. We already know that the characters \n in a string mean a linefeed. However, there are several other important combinations of characters in a string, and some are special to procedures like WriteF. One such combination is \d, which is easier to describe after we’ve seen the changed example.

```
     PROC main()
       WriteF('My first program\n')
       fred()
     ENDPROC

     PROC fred()
       WriteF('...brought to you by the number \d\n', 236)
     ENDPROC
```

You might be able to guess what happens, but compile it and try it out anyway. If everything’s worked you should see that the second message prints out the number that was passed as the second parameter to WriteF. That’s what the \d combination does--it marks the place in the string where the number should be printed. Here’s the output the example should generate:

```
My first program
...brought to you by the number 236
```

Try this next change:

```
PROC main()
  WriteF('My first program\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...the number \d is quite nice\n', 16)
ENDPROC
```

This is very similar, and just shows that the \d really does mark the place where the number is printed. Again, here's the output it should generate:

```
My first program
...the number 16 is quite nice
```

We'll now try printing two numbers.

```
PROC main()
  WriteF('My first program\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...brought to you by the numbers \d and \d\n', 16, 236)
ENDPROC
```

Because we're printing two numbers we need two lots of \d, and we need to supply two numbers as parameters in the order in which we want them to be printed. The number 16 will therefore be printed before the word 'and' and before the number 236. Here's the output:

```
My first program
...brought to you by the numbers 16 and 236
```

We can now make a big step forward and pass the numbers as parameters to the procedure fred. Just look at the differences between this next example and the previous one.

```
PROC main()
  WriteF('My first program\n')
  fred(16, 236)
ENDPROC

PROC fred(a,b)
  WriteF('...brought to you by the numbers \d and \d\n', a,b)
ENDPROC
```

This time we pass the (local) variables a and b to WriteF. This is exactly the same as passing the values they store (which is what the previous example did), and so the output will be the same. In the next section we'll manipulate the variables by doing some arithmetic with a and b, and get WriteF to print the results.

1.26 beginner.guide/Expressions

```
                Expressions
===========
```

The E language includes the normal mathematical and logical operators. These operators are combined with values (usually in variables) to give expressions which yield new values. The following sections discuss this topic in more detail.

Mathematics

Logic and comparison

Precedence and grouping

1.27 beginner.guide/Mathematics

```
                Mathematics
-----------
```

All the standard mathematical operators are supported in E. You can do addition, subtraction, multiplication and division. Other functions such as sine, modulus and square-root can also be used as they are part of the Amiga system libraries, but we only need to know about simple mathematics at the moment. The + character is used for addition, - for subtraction, * for multiplication (it’s the closest you can get to a multiplication sign on a keyboard without using the letter x), and / for division (be careful not to confuse the \ used in strings with / used for division). The following are examples of expressions:

```
       1+2+3+4
       15-5
       5*2
       330/33
       -10+20
       3*3+1
```

Each of these expressions yields ten as its result. The last example is very carefully written to get the precedence correct (see

Precedence and grouping
).

All the above expressions use integer operators, so they manipulate integers, giving integers as results. Floating-point numbers are also supported by E, but using them is quite complicated (see

Floating-Point Numbers
). (Floating-point numbers can represent both very small fractions and very large integers, but they have a limited accuracy, i.e., a limited number of significant digits.)

1.28 beginner.guide/Logic and comparison

Logic and comparison

Logic lies at the very heart of a computer. They rarely guess what to do next; instead they rely on hard facts and precise reasoning. Consider the password protection on most games. The computer must decide whether you entered the correct number or word before it lets you play the game. When you play the game it's constantly making decisions: did your laser hit the alien, have you got any lives left, etc. Logic controls the operation of a program.

In E, the constants TRUE and FALSE represent the truth values true and false (respectively), and the operators AND and OR are the standard logic operators. The comparison operators are = (equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to) and <> (not equal to). All the following expressions are true:

```
TRUE
TRUE AND TRUE
TRUE OR FALSE
1=1
2>1
3<>0
```

And these are all false:

```
FALSE
TRUE AND FALSE
FALSE OR FALSE
0=2
2<1
(2<1) AND (-1=0)
```

The last example must use parentheses. We'll see why in the next section (it's to do with precedence, again).

The truth values TRUE and FALSE are actually numbers. This is how the logic system works in E. TRUE is the number -1 and FALSE is zero. The logic operators AND and OR expect such numbers as their parameters. In fact, the AND and OR operators are really bit-wise operators (see

Bitwise AND and OR
), so most of the time any non-zero number is taken to be TRUE. It can sometimes be convenient to rely on this knowledge,

although most of the time it is preferable (and more readable) to use a slightly more explicit form. Also, these facts can cause a few subtle problems as we shall see in the next section.

1.29 beginner.guide/Precedence and grouping

Precedence and grouping

At school most of us are taught that multiplications must be done before additions in a sum. In E it's different--there is no operator precedence. This means that expressions like 1+3*3 do not give the results a mathematician might expect. In fact, 1+3*3 represents the number 12 in E. This is because the addition, 1+3, is done before the multiplication, since it occurs before the multiplication. If the multiplication were written before the addition it would be done first (like we would normally expect). Therefore, 3*3+1 represents the number 10 in E and in school mathematics.

To overcome this difference we can use parentheses to group the expression. If we'd written 1+(3*3) the result would be 10. This is because we've forced E to do the multiplication first. Although this may seem troublesome to begin with, it's actually a lot better than learning a lot of rules for deciding which operator is done first (in C this can be a real pain, and you usually end up writing the brackets in just to be sure!).

The logic examples above contained the expression:

```
(2<1) AND (-1=0)
```

This expression was false. If we'd left the parentheses out, E would have seen it as:

```
((2<1) AND -1) = 0
```

Now the number -1 shouldn't really be used to represent a truth value with AND, but we do know that TRUE is the number -1, so E will make sense of this and the E compiler won't complain. We will soon see how AND and OR really work (see Bitwise AND and OR), but for now we'll just work out what E would calculate for this expression:

1. Two is not less than one so 2<1 can be replaced by FALSE.

   ```
   (FALSE AND -1) = 0
   ```

2. TRUE is -1 so we can replace -1 by TRUE.

   ```
   (FALSE AND TRUE) = 0
   ```

3. FALSE AND TRUE is FALSE.

```
(FALSE) = 0
```

4. FALSE is really the number zero, so we can replace it with zero.

```
0 = 0
```

5. Zero is equal to zero, so the expression is TRUE.

```
TRUE
```

So E calculates the expression to be true. But the original expression (with parentheses) was false. Bracketing is therefore very important! It is also very easy to do correctly.

1.30 beginner.guide/Program Flow Control

Program Flow Control
****

A computer program often needs to repeatedly execute a series of statements or execute different statements according to the result of some decision. For example, a program to print all the numbers between one and a thousand would be very long and tedious to write if each print statement had to be given individually--it would be much better to use a variable and repeatedly print its value and increment it. Also, things sometimes go wrong and a program must decide whether to continue or print an error message and stop--this part of a program is a typical example of a conditional block.

Conditional Block

Loops

1.31 beginner.guide/Conditional Block

Conditional Block
=================

There are two kinds of conditional block: IF and SELECT. Examples of these blocks are given below as fragments of E code (i.e., the examples are not complete E programs).

```
IF x>0
  x:=x+1
  WriteF('Increment: x is now \d\n', x)
ELSEIF x<0
```

```
         x:=x-1
         WriteF('Decrement: x is now \d\n', x)
       ELSE
         WriteF('Zero: x is 0\n')
       ENDIF
```

In the above IF block, the first part checks if the value of x is greater than zero, and, if it is, x is incremented and the new value is printed (with a message saying it was incremented). The program will then skip the rest of the block, and will execute the statements which follow the ENDIF. If, however, x it is not greater than zero the ELSEIF part is checked, so if x is less than zero it will be decremented and printed, and the rest of the block is skipped. If x is not greater than zero and not less than zero the statements in the ELSE part are executed, so a message saying x is zero is printed. The IF conditional is described in more detail below.

IF block

IF expression

```
                       SELECT x
         CASE 0
           WriteF('x is zero\n')
         CASE 10
           WriteF('x is ten\n')
         CASE -2
           WriteF('x is -2\n')
         DEFAULT
           WriteF('x is not zero, ten or -2\n')
       ENDSELECT
```

The SELECT block is similar to the IF block--it does different things depending on the value of x. However, x is only checked against specific values, given in the series of CASE statements. If it is not any of these values the DEFAULT part is executed.

There's also a variation on the SELECT block (known as the SELECT..OF block) which matches ranges of values and is quite fast. The two kinds of SELECT block are described in more detail below.

SELECT block

SELECT..OF block

1.32 beginner.guide/IF block

IF block

The IF block has the following form (the bits like expression are descriptions of the kinds of E code which is allowed at that point--they are not proper E code):

```
       IF expressionA
         statementsA
       ELSEIF expressionB
         statementsB
       ELSE
         statementsC
       ENDIF
```

This block means:

- If expressionA is true (i.e., represents TRUE or any non-zero number) the code denoted by statementsA is executed.
- If expressionA is false (i.e., represents FALSE or zero) and expressionB is true the statementsB part is executed.
- If both expressionA and expressionB are false the statementsC part is executed.

There does not need to be an ELSE part but if one is present it must be the last part (immediately before the ENDIF). Also, there can be any number of ELSEIF parts between the IF and ELSE parts.

An alternative to this vertical form (where each part is on a separate line) is the horizontal form:

```
       IF expression THEN statementA ELSE statementB
```

This has the disadvantage of no ELSEIF parts and having to cram everything onto a single line. Notice the presence of the THEN keyword to separate the expression and statement. This horizontal form is closely related to the IF expression, which is described below (see
IF expression
).

To help make things clearer here are a number of E code fragments which illustrate the allowable IF blocks:

```
       IF x>0 THEN x:=x+1 ELSE x:=0

       IF x>0
         x:=x+1
       ELSE
         x:=0
       ENDIF

       IF x=0 THEN WriteF('x is zero\n')

       IF x=0
         WriteF('x is zero\n')
       ENDIF

       IF x<0
         Write('Negative x\n')
```

```
       ELSEIF x>2000
         Write('Too big x\n')
       ELSEIF (x=2000) OR (x=0)
         Write('Worrying x\n')
       ENDIF

       IF x>0
         IF x>2000
           WriteF('Big x\n')
         ELSE
           WriteF('OK x\n')
         ENDIF
       ELSE
         IF x<-800 THEN WriteF('Small x\n') ELSE Write('Negative OK x')
       ENDIF
```

In the last example there are nested IF blocks (i.e., an IF block within an IF block). There is no ambiguity in which ELSE or ELSEIF parts belong to which IF block because the beginning and end of the IF blocks are clearly marked. For instance, the first ELSE line can only be interpreted as being part of the innermost IF block.

As a matter of style the conditions on the IF and ELSEIF parts should not overlap (i.e., at most one of the conditions should be true). If they do, however, the first one will take precedence. Therefore, the following two fragments of E code do the same thing:

```
       IF x>0
         WriteF('x is bigger than zero\n')
       ELSEIF x>200
         WriteF('x is bigger than 200\n')
       ELSE
         WriteF('x is too small\n')
       ENDIF

       IF x>0
         WriteF('x is bigger than zero\n')
       ELSE
         WriteF('x is too small\n')
       ENDIF
```

The ELSEIF part of the first fragment checks whether x is greater than 200. But, if it is, the check in the IF part would have been true (x is certainly greater than zero if it's greater than 200), and so only the code in the IF part is executed. The whole IF block behaves as if the ELSEIF was not there.

1.33 beginner.guide/IF expression

IF expression

IF is such a commonly used construction that there is also an IF expression. The IF block is a statement and it controls which lines of code are executed, whereas the IF expression is an expression and it controls its own value. For example, the following IF block:

```
IF x>0
  y:=x+1
ELSE
  y:=0
ENDIF
```

can be written more succinctly using an IF expression:

```
y:=(IF x>0 THEN x+1 ELSE 0)
```

The parentheses are unnecessary but they help to make the example more readable. Since the IF block is just choosing between two assignments to y it isn’t really the lines of code that are different (they are both assignments), rather it is the values that are assigned to y that are different. The IF expression makes this similarity very clear. It chooses the value to be assigned in just the same way that the IF block choose the assignment.

As you can see, IF expressions are written like the horizontal form of the IF block. However, there must be an ELSE part and there can be no ELSEIF parts. This means that the expression will always have a value, and it isn’t cluttered with lots of cases.

Don’t worry too much about IF expressions, since there are only useful in a handful of cases and can always be rewritten as a more wordy IF block. Having said that they are very elegant and a lot more readable than the equivalent IF block.

1.34 beginner.guide/SELECT block

SELECT block

The SELECT block has the following form:

```
SELECT variable
CASE expressionA
  statementsA
CASE expressionB
  statementsB
DEFAULT
  statementsC
ENDSELECT
```

The value of the selection variable (denoted by variable in the SELECT part) is compared with the value of the expression in each of the CASE parts in turn. If there’s a match, the statements in the (first) matching CASE part are executed. There can be any number of CASE parts between the SELECT and DEFAULT parts. If there is no match, the statements in the DEFAULT part are executed. There does not need to be a DEFAULT part but if one is present it must be the last part (immediately before the ENDSELECT).

It should be clear that SELECT blocks can be rewritten as IF blocks, with the checks on the IF and ELSEIF parts being equality checks. For example, the following code fragments are equivalent:

```
SELECT x
CASE 22
  WriteF('x is 22\n')
CASE (y+z)/2
  WriteF('x is (y+x)/2\n')
DEFAULT
  WriteF('x isn't anything significant\n')
ENDSELECT

IF x=22
  WriteF('x is 22\n')
ELSEIF x=(y+z)/2
  WriteF('x is (y+x)/2\n')
ELSE
  WriteF('x isn't anything significant\n')
ENDIF
```

Notice that the IF and ELSEIF parts come from the CASE parts, the ELSE part comes from the DEFAULT part, and the order of the parts is preserved. The advantage of the SELECT block is that it's much easier to see that the value of x is being tested all the time, and also we don't have to keep writing x= in the checks.

1.35 beginner.guide/SELECT..OF block

SELECT..OF block

The SELECT..OF block is a bit more complicated than the normal SELECT block, but can be very useful. It has the following form:

```
SELECT maxrange OF expression
CASE constA
  statementsA
CASE constB1 TO constB2
  statementsB
CASE range1, range2
  statementsC
DEFAULT
  statementsD
ENDSELECT
```

The value to be matched is expression, which can be any expression, not just a variable like in the normal SELECT block. However, the maxrange, constA, constB1 and constB2 must all be explicit numbers, i.e., constants (see
Constants

). maxrange must be a positive constant and the other constants must be between zero and maxrange (including zero but excluding maxrange).

The CASE values to be matched are specified using ranges. A simple range is a single constant (the first CASE above). The more general range is shown in the second CASE, using the TO keyword (constB2 must be greater than constB1). A general CASE in the SELECT..OF block can specify a number of possible ranges to match against by separating each range with a comma, as in the third CASE above. For example, the following CASE lines are equivalent and can be used to match any number from one to five (inclusive):

```
CASE 1 TO 5

CASE 1, 2, 3, 4, 5

CASE 1 TO 3, 3 TO 5

CASE 1, 2 TO 3, 4, 5

CASE 1, 5, 2, 4, 3

CASE 2 TO 3, 5, 1, 4
```

If the value of the expression is less than zero, greater than or equal to maxrange, or it does not match any of the constants in the CASE ranges, then the statements in the DEFAULT part are executed. Otherwise the statements in the first matching CASE part are executed. As in the normal SELECT block, there does not need to be a DEFAULT part.

The following SELECT..OF block prints the (numeric) day of the month nicely:

```
SELECT 32 OF day
CASE 1, 21, 31
  WriteF('The \dst day of the month\n', day)
CASE 2, 22
  WriteF('The \dnd day of the month\n', day)
CASE 3, 23
  WriteF('The \drd day of the month\n', day)
CASE 4 TO 20, 24 TO 30
  WriteF('The \dth day of the month\n', day)
DEFAULT
  WriteF('Error: invalid day=\d\n', day)
ENDSELECT
```

The maxrange for this block is 32, since 31 is the maximum of the values used in the CASE parts. If the value of day was 100, for instance, then the statements in the DEFAULT part would be executed, signalling an invalid day.

This example can be rewritten as an IF block:

```
IF (day=1) OR (day=21) OR (day=31)
  WriteF('The \dst day of the month\n', day)
ELSEIF (day=2) OR (day=22)
```

```
          WriteF('The \dnd day of the month\n', day)
        ELSEIF (day=3) OR (day=23)
          WriteF('The \drd day of the month\n', day)
        ELSEIF ((4<=day) AND (day<=20))  OR  ((24<=day) AND (day<=30))
          WriteF('The \dth day of the month\n', day)
        ELSE
          WriteF('Error: invalid day=\d\n', day)
        ENDIF
```

The comma separating two ranges in the CASE part has been replaced by an OR of two comparison expressions, and the TO range has been replaced by an AND of two comparisons. (It is worth noticing the careful bracketing of the resulting expressions.)

Clearly, the SELECT..OF block is much more readable than the equivalent IF block. It is also a lot faster, mainly because none of the comparisons present in IF block have to be done in the SELECT..OF version. Instead the value to be matched is used to immediately locate the correct CASE part. However, it's not all good news: the maxrange value directly affects the size of compiled executable, so it is recommended that SELECT..OF blocks be used only with small maxrange values. See the 'Reference Manual' for more details.

1.36 beginner.guide/Loops

Loops
=====

Loops are all about making a program execute a series of statements over and over again. Probably the simplest loop to understand is the FOR loop. There are other kinds of loops, but they are easier to understand once we know how to use a FOR loop.

FOR loop

WHILE loop

REPEAT..UNTIL loop

1.37 beginner.guide/FOR loop

FOR loop

If you want to write a program to print the numbers one to 100 you can either type each number and wear out your fingers, or you can use a single variable and a small FOR loop. Try compiling this E program (the space after the \d in the string is needed to separate the printed numbers):

```
PROC main()
  DEF x
  FOR x:=1 TO 100
    WriteF('\d ', x)
  ENDFOR
  WriteF('\n')
ENDPROC
```

When you run this you'll get all the numbers from one to 100 printed, just like we wanted. It works by using the (local) variable x to hold the number to be printed. The FOR loop starts off by setting the value of x to one (the bit that looks like an assignment). Then the statements between the FOR and ENDFOR lines are executed (so the value of x gets printed). When the program reaches the ENDFOR it increments x and checks to see if it is bigger than 100 (the limit we set with the TO part). If it is, the loop is finished and the statements after the ENDFOR are executed. If, however, it wasn't bigger than 100, the statements between the FOR and ENDFOR lines are executed all over again, and this time x is one bigger since it has been incremented. In fact, this program does exactly the same as the following program (the ... is not E code--it stands for the 97 other WriteF statements):

```
PROC main()
  WriteF('\d ', 1)
  WriteF('\d ', 2)
  ...
  WriteF('\d ', 100)
  WriteF('\n')
ENDPROC
```

The general form of the FOR loop is as follows:

```
FOR var := expressionA TO expressionB STEP number
  statements
ENDFOR
```

The var bit stands for the loop variable (in the example above this was x). The expressionA bit gives the start value for the loop variable and the expressionB bit gives the last allowable value for it. The STEP part allows you to specify the value (given by number) which is added to the loop variable on each loop. Unlike the values given for the start and end (which can be arbitrary expressions), the STEP value must be a constant (see Constants). The STEP value defaults to one if the STEP part is omitted (as in our example). Negative STEP values are allowed, but in this case the check used at the end of each loop is whether the loop variable is less than the value in the TO part. Zero is not allowed as the STEP value.

As with the IF block there is a horizontal form of a FOR loop:

```
FOR var := expA TO expB STEP expC DO statement
```

1.38 beginner.guide/WHILE loop

WHILE loop

The FOR loop used a loop variable and checked whether that variable had gone past its limit. A WHILE loop allows you to specify your own loop check. For instance, this program does the same as the program in the previous section:

```
PROC main()
  DEF x
  x:=1
  WHILE x<=100
    WriteF('\d ', x)
    x:=x+1
  ENDWHILE
  WriteF('\n')
ENDPROC
```

We've replaced the FOR loop with an initialisation of x and a WHILE loop with an extra statement to increment x. We can now see the inner workings of the FOR loop and, in fact, this is exactly how the FOR loop works.

It is important to know that our check, x<=100, is done before the loop statements are executed. This means that the loop statements might not even be executed once. For instance, if we'd made the check x>=100 it would be false at the beginning of the loop (since x is initialised to one in the assignment before the loop). Therefore, the loop would have terminated immediately and execution would pass straight to the statements after the ENDWHILE.

Here's a more complicated example:

```
PROC main()
  DEF x,y
  x:=1
  y:=2
  WHILE (x<10) AND (y<10)
    WriteF('x is \d and y is \d\n', x, y)
    x:=x+2
    y:=y+2
  ENDWHILE
ENDPROC
```

We've used two (local) variables this time. As soon as one of them is ten or more the loop is terminated. A bit of inspection of the code reveals that x is initialised to one, and keeps having two added to it. It will, therefore, always be an odd number. Similarly, y will always be even. The WHILE check shows that it won't print any numbers which are greater than or equal to ten. From this and the fact that x starts at one and y at two we can decide that the last pair of numbers will be seven and eight. Run the program to confirm this. It should produce the following output:

```
x is 1 and y is 2
x is 3 and y is 4
x is 5 and y is 6
x is 7 and y is 8
```

Like the FOR loop, there is a horizontal form of the WHILE loop:

```
WHILE expression DO statement
```

Loop termination is always a big problem. FOR loops are guaranteed to eventually reach their limit (if you don't mess with the loop variable, that is). However, WHILE loops (and all other loops) may go on forever and never terminate. For example, if the loop check were 1<2 it would always be true and nothing the loop could do would prevent it being true! You must therefore take care that you make sure your loops terminate in some way if you want to program to finish. There is a sneaky way of terminating loops using the JUMP statement, but we'll ignore that for now.

1.39 beginner.guide/REPEAT..UNTIL loop

REPEAT..UNTIL loop

A REPEAT..UNTIL loop is very similar to a WHILE loop. The only difference is where you specify the loop check, and when and how the check is performed. To illustrate this, here's the program from the previous two sections rewritten using a REPEAT..UNTIL loop (try to spot the subtle differences):

```
PROC main()
  DEF x
  x:=1
  REPEAT
    WriteF('\d ', x)
    x:=x+1
  UNTIL x>100
  WriteF('\n')
ENDPROC
```

Just as in the WHILE loop version we've got an initialisation of x and an extra statement in the loop to increment x. However, this time the loop check is specified at the end of the loop (in the UNTIL part), and the check is only performed at the end of each loop. This difference means that the code in a REPEAT..UNTIL loop will be executed at least once, whereas the code in a WHILE loop may never be executed. Also, the logical sense of the check follows the English: a REPEAT..UNTIL loop executes until the check is true, whereas the WHILE loop executes while the check is true. Therefore, the REPEAT..UNTIL loop executes while the check is false! This may seem confusing at first, but just remember to read the code as if it were English and you'll get the correct interpretation.

1.40 beginner.guide/Summary

Summary

This is the end of Part One, which was hopefully enough to get you started. If you’ve grasped the main concepts you are good position to attack Part Two, which covers the E language in more detail.

This is probably a good time to look at the different parts of one of the examples from the previous sections, since we’ve now used quite a bit of E. The following examination uses the WHILE loop example. Just to make things easier to follow, each line has been numbered (don’t try to compile it with the line numbers on!).

```
 1.  PROC main()
 2.    DEF x,y
 3.    x:=1
 4.    y:=2
 5.    WHILE (x<10) AND (y<10)
 6.      WriteF('x is \d and y is \d\n', x, y)
 7.      x:=x+2
 8.      y:=y+2
 9.    ENDWHILE
10.  ENDPROC
```

Hopefully, you should be able to recognise all the features listed in the table below. If you don’t then you might need to go back over the previous chapters, or find a much better programming guide than this!

Line(s)	Observation
1-10	The procedure definition.
1	The declaration of the procedure main, with no parameters.
2	The declaration of local variables x and y.
3, 4	Initialisation of x and y using assignment statements.
5-9	The WHILE loop.
5	The loop check for the WHILE loop using the logical operator AND, the comparison operator <, and parentheses to group the expression.
6	The call to the (built-in) procedure WriteF using parameters. Notice the string, the place holders for numbers, \d, and the linefeed, \n.
7, 8	Assignments to x and y, adding two to their values.

9 The marker for the end of the WHILE loop.

10 The marker for the end of the procedure.

1.41 beginner.guide/Format and Layout

Format and Layout
****

In this chapter we'll look at the rules which govern the format and layout of E code. In the previous Part we saw examples of E code that were quite nicely indented and the structure of the program was easily visible. This was just a convention and the E language does not constrain you to write code in this way. However, there are certain rules that must be followed. (This chapter refers to some concepts and parts of the E language which were not covered in Part One. Don't let this put you off--those things will be dealt with in later chapters, and it's maybe a good idea to read this chapter again when they have been.)

Identifiers

Statements

Spacing and Separators

Comments

1.42 beginner.guide/Identifiers

Identifiers
===========

An identifier is a word which the compiler must interpret rather than treating literally. For instance, a variable is an identifier, as is a keyword (e.g., IF), but anything in a string is not (e.g., fred in 'fred and wilma' is not an identifier). Identifiers can be made up of upper- or lower-case letters, numbers and underscores (the _ character). There are only two constraints:

1. The first character cannot be a number (this would cause confusion with numeric constants).

2. The case of the first few characters of identifiers is significant.

For keywords (e.g., ENDPROC), constants (e.g., TRUE) and assembly mnemonics (e.g., MOVE.L) the first two characters must both be uppercase. For E built-in or Amiga system procedures/functions the first character must be uppercase and the second must be lowercase. For all other identifiers (i.e., local, global and procedure parameter variables, object names and element names, procedure names and code labels) the first character must be lowercase.

Apart from these constraints you are free to write identifiers how you like, although it's arguably more tasteful to use all lowercase for variables and all uppercase for keywords and constants.

1.43 beginner.guide/Statements

Statements
==========

A statement is normally a single line of an instruction to the computer. Each statement normally occupies a single line. If a procedure is thought of as a paragraph then a statement is a sentence. Variables, expressions and keywords are the words which make up the sentence.

So far in our examples we have met only two kinds of statement: the single line statement and the multi-line statement. The assignments we have seen were single line statements, and the vertical form of the IF block is a multi-line statement. The horizontal form of the IF block was actually the single line statement form of the IF block. Notice that statements can be built up from other statements, as is the case for IF blocks. The code parts between the IF, ELSEIF, ELSE and ENDIF lines are sequences of statements.

Single line statements can often be very short, and you may be able to fit several of them onto an single line without the line getting too long. To do this in E you use a semi-colon (the ; character) to separate each statement on the line. For example, the following code fragments are equivalent:

```
fred(y,z)
y:=x
x:=z+1

fred(y,z); y:=x; x:=z+1
```

On the other hand you may want to split a long statement over several lines. This is a bit more tricky because the compiler needs to see that you haven't finished the statement when it gets to the end of a line. Therefore you can only break a statement at certain places. The most common place is after a comma that is part of the statement (like in a procedure call with more than one parameter), but you can also split a line after binary operators and anywhere between opening and closing brackets. The following examples are rather silly but show some allowable line breaking places.

```
fred(a, b, c,
     d, e, f)   /* After a comma */

x:=x+
   y+
```

```
           z            /* After a binary operator */

        x:=(1+2
            +3)         /* Between open...close brackets */

        list:= [ 1,2,
                 [3,4],
                ]       /* Between open...close brackets */
```

The simple rule is this: if a complete line can be interpreted as a statement then it will be, otherwise it will be interpreted as part of a statement which continues on the following lines.

Strings may also get a bit long. You can split them over several lines by breaking them into several separate strings and using + between them. If a line ends with a + and the previous thing on the line was a string then the E compiler takes the next string to be a continuation. The following calls to WriteF print the same thing:

```
       WriteF(’This long string can be broken over several lines.\n’)

       WriteF(’This long string ’ +
              ’can be broken over several lines.\n’)

       WriteF(’This long’ +
              ’ string can be ’ +
              ’broken over several ’ +
              ’lines.\n’)
```

1.44 beginner.guide/Spacing and Separators

Spacing and Separators

The examples we’ve seen so far used a rigid indentation convention which was intended to illuminate the structure of the program. This was just a convention, and the E language places no constraints on the amount of whitespace (spaces, tabs and linefeeds) you place between statements. However, within statements you must supply enough spacing to make the statement readable. This generally means that you must put whitespace between adjacent identifiers which start or end with a letter, number or underscore (so that the compiler does not think it’s one big identifier!). In practice this means you should put a space after a keyword if it might run into a variable or procedure name. Most other times (like in expressions) identifiers are separated by non-identifier characters (a comma, parenthesis or other symbol).

1.45 beginner.guide/Comments

Comments
========

A comment is something that the E compiler ignores and is only there to help the reader. Remember that one day in the future you may be the reader, and it may be quite hard to decipher your own code without a few decent comments! Comments are therefore pretty important.

You can write comments anywhere you can write whitespace that isn’t part of a string. There are two kinds of comment: one uses /* to mark the start of the comment text and */ to mark the end, and the other uses -> to mark the start, with the comment text continuing to the end of the line. You must be careful not to write /*, */ or -> as part of the comment text, unless part of a nested comment. In practice a comment is best put on a line by itself or after the end of the code on a line.

```
        /* This line is a comment */
        x:=1  /* This line contains an assignment then a comment */
     /* y:=2  /* This whole line is a comment with a nested comment */*/

        x:=1  -> Assignment then a comment
     -> y:=2  /* A nested comment comment */
```

1.46 beginner.guide/Procedures and Functions

Procedures and Functions
****

A function is a procedure which returns a value. This value can be any expression so it may depend on the parameters with which the function was called. For instance, the addition operator + can be thought of as a function which returns the sum of its two parameters.

Functions

One-Line Functions

Default Arguments

Multiple Return Values

1.47 beginner.guide/Functions

Functions
=========

We can define our own addition function, add, in a very similar way to the definition of a procedure. (The only difference is that a function explicitly returns a value.)

```
PROC main()
  DEF sum
  sum:=12+79
  WriteF('Using +, sum is \d\n', sum)
  sum:=add(12,79)
  WriteF('Using add, sum is \d\n', sum)
ENDPROC

PROC add(x, y)
  DEF s
  s:=x+y
ENDPROC s
```

This should generate the following output:

```
Using +, sum is 91
Using add, sum is 91
```

In the procedure add the value s is returned using the ENDPROC label. The value returned from add can be used in expressions, just like any other value. You do this by writing the procedure call where you want the value to be. In the above example we wanted the value to be assigned to sum so we wrote the call to add on the right-hand side of the assignment. Notice the similarities between the uses of + and add. In general, add(a,b) can be used in exactly the same places that a+b can (more precisely, it can be used anywhere (a+b) can be used).

The RETURN keyword can also be used to return values from a procedure. If the ENDPROC method is used then the value is returned when the procedure reaches the end of its code. However, if the RETURN method is used the value is returned immediately at that point and no more of the procedure's code is executed. Here's the same example using RETURN:

```
PROC add(x, y)
  DEF s
  s:=x+y
  RETURN s
ENDPROC
```

The only difference is that you can write RETURN anywhere in the code part of a procedure and it finishes the execution of the procedure at that point (rather than execution finishing when it reaches the end of the code). In fact, you can use RETURN in the main procedure to prematurely finish the execution of a program.

Here's a slightly more complicated use of RETURN:

```
PROC limitedadd(x,y)
  IF x>10000
    RETURN 10000
  ELSEIF x<-10000
    RETURN -10000
  ELSE
    RETURN x+y
```

```
    ENDIF
    /* The following code is redundant */
    x:=1
    IF x=1 THEN RETURN 9999 ELSE RETURN -9999
  ENDPROC
```

This function checks to see if x is greater than 10,000 or less than -10,000, and if it is a limited value is returned (which is generally not the correct sum!). If x is between -10,000 and 10,000 the correct answer is returned. The lines after the first IF block will never get executed because execution will have finished at one of the RETURN lines. Those lines are therefore just a waste of compiler time and can safely be omitted (as the comment suggests).

If no value is given with the ENDPROC or RETURN keyword then zero is returned. Therefore, all procedures are actually functions (and the terms procedure and function will tend to be used interchangeably). So, what happens to the value when you write a procedure call on a line by itself, not in an expression? Well, as we will see, the value is simply discarded (see Turning an Expression into a Statement). This is what happened in the previous examples when we called the procedures fred and WriteF.

1.48 beginner.guide/One-Line Functions

One-Line Functions
==================

Just as the IF block and FOR loop have horizontal, single line forms, so does a procedure definition. The general form is:

```
PROC name (arg1, arg2, ...) IS expression
```

Alternatively, the RETURN keyword can be used:

```
PROC name (arg1, arg2, ...) RETURN expression
```

At first sight this might seem pretty unusable, but it is useful for very simple functions and our add function in the previous section is a good example. If you look closely at the original definition you'll see that the local variable s wasn't really needed. Here's the one-line definition of add:

```
PROC add(x,y) IS x+y
```

1.49 beginner.guide/Default Arguments

Default Arguments
=================

Sometimes a procedure (or function) will quite often be called with a particular (constant) value for one of its parameters, and it might be nice if you didn't have to fill this value in all the time. Luckily, E allows you to define default values for a procedure's parameters when you define the procedure. You can then just leave out that parameter when you call the procedure and it will default to the value you defined for it. Here's a simple example:

```
PROC play(track=1)
  WriteF('Starting to play track \d\n', track)
  /* Rest of the code... */
ENDPROC

PROC main()
  play(1)  -> Start playing from track 1
  play(6)  -> Start playing from track 6
  play()   -> Start playing from track 1
ENDPROC
```

This is an outline of a program to control something like a CD player. The play procedure has one parameter, track, which represents the first track that should be played. Often, though, you just tell the CD player to play, and don't specify a particular track. In this case, play starts from the first track. This is exactly what happens in the example above: the track parameter has a default value of 1 defined for it (the =1 in the definition of the play procedure), and the third call to play in main does not specify a value for track, so the default value is used.

There are two constraints on the use of default arguments:

1. Any number of the parameters of a procedure may have default values defined for them, although they may only be the right-most parameters. This means that for a three parameter procedure, the second parameter can have a default value only if the last parameter does as well, and the first can have one only if both the others do. This should not be a big problem because you can always reorder the parameters in the procedure definition.

   The following examples show legal definitions of procedures with default arguments:

   ```
   PROC fred(x, y, z) IS x+y+z         -> No defaults

   PROC fred(x, y, z=1) IS x+y+z       -> z defaults to 1

   PROC fred(x, y=23, z=1) IS x+y+z    -> y and z have defaults

   PROC fred(x=9, y=23, z=1) IS x+y+z  -> All have defaults
   ```

   On the other hand, these definitions are all illegal:

   ```
   PROC fred(x, y=23, z) IS x+y+z   -> Illegal: no z default
   ```

```
PROC fred(x=9, y, z=1) IS x+y+z  -> Illegal: no y default
```

2. When you call a procedure which has default arguments you can only leave out the right-most parameters. This means that for a three parameter procedure with all three parameters having default values, you can leave out the second parameter in a call to this procedure only if you also leave out the third parameter. The first parameter may be left out only if both the others are, too.

   The following example shows which parameters are considered defaults:

   ```
   PROC fred(x, y=23, z=1)
     WriteF('x is \d, y is \d, z is \d\n', x, y, z)
   ENDPROC

   PROC main()
     fred(2, 3, 4) -> No defaults used
     fred(2, 3)    -> z defaults to 1
     fred(2)       -> y and z default
     fred()        -> Illegal: x has no default
   ENDPROC
   ```

   In this example, you cannot leave out the y parameter in a call to fred without leaving out the z parameter as well. To make y have its default value and z some value other than its default you need to supply the y value explicitly in the call:

   ```
     fred(2, 23, 9) -> Need to supply 23 for y
   ```

These constraints are necessary in order to make procedure calls unambiguous. Consider a three-parameter procedure with default values for two of the parameters. If it is called with only two parameters then, without these constraints, it would not be clear which two parameters had been supplied and which had not. If, however, the procedure were defined and called according to these constraints, then it must be the third parameter that needs to be defaulted (and the two parameters with default values must be the last two).

1.50 beginner.guide/Multiple Return Values

Multiple Return Values

So far we've only seen functions which return only one value, since this is something common to most programming languages. However, E allows you to return up to three values from a function. To do this you list the values separated by commas after the ENDPROC, RETURN or IS keyword, where you would normally have specified only one value. A good example is a function which manipulates a screen coordinate, which is a pair of values: the x- and y-coordinates.

```
PROC movediag(x, y) IS x+8, y+4
```

All this function does is add 8 to the x-coordinate and 4 to the y-coordinate. To get to the return values other than the first one you must use a multiple-assignment statement:

```
     PROC main()
       DEF a, b
       a, b:=movediag(10, 3)
       /* Now a should be 10+8, and b should be 3+4 */
       WriteF('a is \d, b is \d\n', a, b)
     ENDPROC
```

a is assigned the first return value and b is assigned the second. You don't need to use all the return values from a function, so the assignment in the example above could have assigned only to a (in which case it would not be a multiple-assignment anymore). A multiple-assignment makes sense only if the right-hand side is a function call, so don't expect things like the following example to set b properly:

```
       a,b:=6+movediag(10,3)  -> No obvious value for b
```

If you use a function with more than one return value in any other expression (i.e., something which is not the right-hand side of an assignment), then only the first return value is used. For this reason the return values of a function have special names: the first return value is called the regular value of the function, and the other values are the optional values.

```
     PROC main()
       DEF a, b
       /* The next two lines ignore the second return value */
       a:=movediag(10, 3)
       WriteF('x-coord of movediag(21, 4) is \d\n', movediag(21,4))
     ENDPROC
```

1.51 beginner.guide/Constants

Constants
****

A constant is a value that does not change. A (literal) number like 121 is a good example of a constant--its value is always 121. We've already met another kind of constant: string constants (see Strings). As you can doubtless tell, constants are pretty important things.

Numeric Constants

String Constants Special Character Sequences

Named Constants

Enumerations

Sets

1.52 beginner.guide/Numeric Constants

Numeric Constants

We've met a lot of numbers in the previous examples. Technically speaking, these were numeric constants (constant because they don't change value like a variable might). They were all decimal numbers, but you can use hexadecimal and binary numbers as well. There's also a way of specifying a number using characters. To specify a hexadecimal number you use a $ before the digits (and after the optional minus sign - to represent a negative value). To specify a binary number you use a % instead.

Specifying numbers using characters is more complicated, because the base of this system is 256 (the base of decimal is ten, that of hexadecimal is 16 and that of binary is two). The digits are enclosed in double-quotes (the " character), and there can be at most four digits. Each digit is a character representing its ASCII value. Therefore, the character A represents 65 and the character 0 (zero) represents 48. This upshot of this is that character A has ASCII value "A" in E, and "0z" represents ("0" * 256) + "z" = (48 * 256) + 122 = 12,410. However, you probably don't need to worry about anything other than the single character case, which gives you the ASCII value of the character.

The following table shows the decimal value of several numeric constants. Notice that you can use upper- or lower-case letters for the hexadecimal constants. Obviously the case of characters is significant for character numbers.

Number	Decimal value
21	21
-143	-143
$1a	26
-$B1	-177
%1110	14
-%1010	-10
"z"	122
"Je"	19,045
-"A"	-65

1.53 beginner.guide/String Constants Special Character Sequences

String Constants: Special Character Sequences

We have seen that in a string the character sequence \n means a linefeed (see Strings). There are several other similar such special character sequences which represent useful characters that can't be typed in a string. The following table shows all these sequences. Note that there are some other similar sequences which are used to control formatting with built-in procedures like WriteF. These are listed where WriteF and similar procedures are described (see Input and output functions).

Sequence	Meaning
\0	A null (ASCII zero)
\a	An apostrophe '
\b	A carriage return (ASCII 13)
\e	An escape (ASCII 27)
\n	A linefeed (ASCII 10)
\q	A double quote (ASCII 34)
\t	A tab (ASCII 9)
\	A backslash \

An apostrophe can also be produced by typing two apostrophes in a row in a string. It's best to use this only in the middle of a string, where it's nice and obvious:

```
WriteF('Here\as an apostrophe.\n')       /* Using \a */

WriteF('Here''s another apostrophe.\n')  /* Using '' */
```

1.54 beginner.guide/Named Constants

Named Constants
===============

It is often nice to be able to give names to certain constants. For instance, as we saw earlier, the truth value TRUE actually represents the value -1, and FALSE represents zero (see Logic and comparison). These are our first examples of named constants. To define your own you use the CONST keyword as follows:

```
CONST ONE=1, LINEFEED=10, BIG_NUM=999999
```

This has defined the constant ONE to represent one, LINEFEED ten and BIG_NUM 999,999. Named constants must begin with two uppercase letters, as mentioned before (see Identifiers).

You can use previously defined constants to give the value of a new constant, but in this case the definitions must occur on different CONST lines.

```
CONST ZERO=0
CONST ONE=ZERO+1
CONST TWO=ONE+1
```

The expression used to define the value of a constant can use only simple operators (no function calls) and constants.

1.55 beginner.guide/Enumerations

Enumerations
============

Often you want to define a whole lot of constants and you just want them all to have a different value so you can tell them apart easily. For instance, if you wanted to define some constants to represent some famous cities and you only needed to know how to distinguish one from another then you could use an enumeration like this:

```
ENUM LONDON, MOSCOW, NEW_YORK, PARIS, ROME, TOKYO
```

The ENUM keyword begins the definitions (like the CONST keyword does for an ordinary constant definition). The actual values of the constants start at zero and stretch up to five. In fact, this is exactly the same as writing:

```
CONST LONDON=0, MOSCOW=1, NEW_YORK=2, PARIS=3, ROME=4, TOKYO=5
```

The enumeration does not have to start at zero, though. You can change the starting value at any point by specifying a value for an enumerated constant. For example, the following constant definitions are equivalent:

```
ENUM APPLE, ORANGE, CAT=55, DOG, GOLDFISH, FRED=-2,
     BARNEY, WILMA, BETTY

CONST APPLE=0, ORANGE=1, CAT=55, DOG=56, GOLDFISH=57,
      FRED=-2, BARNEY=-1, WILMA=0, BETTY=1
```

1.56 beginner.guide/Sets

                Sets
====

Yet another kind of constant definition is the set definition. This useful for defining flag sets, i.e., a number of options each of which can be on or off. The definition is like a simple enumeration, but using the

SET keyword and this time the values start at one and increase as powers of two (so the next value is two, the next is four, the next eight, and so on). Therefore, the following definitions are equivalent:

```
SET ENGLISH, FRENCH, GERMAN, JAPANESE, RUSSIAN

CONST ENGLISH=1, FRENCH=2, GERMAN=4, JAPANESE=8, RUSSIAN=16
```

However, the significance of the values it is best shown by using binary constants:

```
CONST ENGLISH=%00001, FRENCH=%00010, GERMAN=%00100,
      JAPANESE=%01000, RUSSIAN=%10000
```

If a person speaks just English then we can use the constant ENGLISH. If they also spoke Japanese then to represent this with a single value we'd normally need a new constant (something like ENG_JAP). In fact, we'd probably need a constant for each combination of languages a person might know. However, with the set definition we can OR the ENGLISH and JAPANESE values together to get a new value, %01001, and this represents a set containing both ENGLISH and JAPANESE. On the other hand, to find out if someone speaks French we would AND the value for the languages they know with %00010 (or the constant FRENCH). (As you might have guessed, AND and OR are really bit-wise operators, not simply logical operators. See

Bitwise AND and OR
.)

Consider this program fragment:

```
speak:=GERMAN OR ENGLISH OR RUSSIAN  /* Speak any of these */
IF speak AND JAPANESE
  WriteF('Can speak in Japanese\n')
ELSE
  WriteF('Unable to speak in Japanese\n')
ENDIF
IF speak AND (GERMAN OR FRENCH)
  WriteF('Can speak in German or French\n')
ELSE
  WriteF('Unable to speak in German or French\n')
ENDIF
```

The assignment sets speak to show that the person can speak in German, English or Russian. The first IF block tests whether the person can speak in Japanese, and the second tests whether they can speak in German or French.

When using sets be careful you don't get tempted to add values instead of OR-ing them. Adding two different constants from the same set is the same as OR-ing them, but adding a constant to itself isn't. This is not the only time addition doesn't give the same answer, but it's the most obvious. If you to stick to using OR you won't have a problem.

1.57 beginner.guide/Types

Types
*****

We've already met the LONG type and found that this was the normal type for variables (see Variable types). The types INT and LIST were also mentioned. Learning how to use types in an effective and readable way is very important. The type of a variable (as well as its name) can give clues to the reader about how or for what it is used. There are also more fundamental reasons for needing types, e.g., to logically group data using objects (see OBJECT Type).

This is a very large chapter and you might like to take it slowly. One of the most important things to get to grips with is pointers. Concentrate on trying to understand these as they play a large part in any kind of system programming.

1.58 beginner.guide/LONG Type

LONG Type
=========

The LONG type is the most important type because it is the default type and by far the most common type. It can be used to store a variety of data, including memory addresses, as we shall see.

1.59 beginner.guide/Default type

Default type

LONG is the default type of variables. It is a 32-bit type, meaning that 32-bits of memory (RAM) are used to store the data for each variable of this type and the data can take (integer) values in the range -2,147,483,648 to 2,147,483,647. Variables default to being LONG typed, but they can also be explicitly declared as LONG:

```
     DEF x:LONG, y

     PROC fred(p:LONG, q, r:LONG)
       DEF zed:LONG
       statements
     ENDPROC
```

The global variable x, procedure parameters p and r, and local variable zed have all been declared to be LONG values. The declarations are very similar to the kinds we’ve seen before, except that the variables have :LONG after their name in the declaration. This is the way the type of a variable is given. Note that the global variable y and the procedure parameter q are also LONG, since they do not have a type specified and LONG is the default type for variables.

1.60 beginner.guide/Memory addresses

Memory addresses

There’s a very good reason why LONG is the normal type. A 32-bit (integer) value can be used as a memory address. Therefore we can store the address (or location) of data in a variable (the variable is then called a pointer). The variable would then not contain the value of the data but a way of finding the data. Once the data location is known the data can be read or even altered! The next section covers pointers and addresses in more detail. See
PTR Type
.

1.61 beginner.guide/PTR Type

PTR Type
========

The PTR type is used to hold memory addresses. Variables which have a PTR type are called pointers (since they store memory addresses, as mentioned in the previous section). This section describes, in detail, addresses, pointers and the PTR type.

Addresses

Pointers

Indirect types

Finding addresses (making pointers)

Extracting data (dereferencing pointers)

Procedure parameters

1.62 beginner.guide/Addresses

Addresses

To understand memory addresses, a good analogy is to think of memory as a road or street, each memory location as a post-box on a house, and each piece of data as a letter. If you were a postman you would need to know where to put your letters, and this information is given by the address of the post-box. As time goes by, each post-box is filled with different letters. This is like the value in a memory location (or variable) changing. To change the letters stored in your post-box, you tell your friends your address and they can send letters in and fill it. This is like letting some program change your data by giving it the address of the data.

The next two diagrams illustrate this analogy. A letter contains an address which points to a particular house (or lot of mail) on a street.

```
            +-------+
            | Letter|
            |-------|
            |Address+----*
            +-------+     \
                           \
                            \
              +--------+ +---\----+ +--------+     +--------+
              | House  | | House  | | House  |     | House  |
     Street:  |+------+| |+------+| |+------+| ... |+------+|
              || Mail || || Mail || || Mail ||     || Mail ||
              +========+ +========+ +========+     +========+
```

A pointer contains an address which points to a variable (or data) in memory.

```
            +-------+
            |Pointer|
            |-------|
```

```
            |Address+----*
            +-------+     \
                           \
                            \
              +--------+ +---\----+ +--------+      +--------+
              |Variable| |Variable| |Variable|      |Variable|
     Memory:  |+------+| |+------+| |+------+| ... |+------+|
              || Data || || Data || || Data ||      || Data ||
              +========+ +========+ +========+      +========+
```

1.63 beginner.guide/Pointers

Pointers

Variables which contain memory addresses are called pointers. As we saw in the previous section, we can store memory addresses in LONG variables. However, we then don't know the type of the data stored at those addresses. If it is important (or useful) to know this then the PTR type (or, more accurately, one of the many PTR types) should be used.

```
     DEF p:PTR TO LONG, i:PTR TO INT,
         cptr:PTR TO CHAR, gptr:PTR TO gadget
```

The values stored in each of p, cptr, i and gptr are LONG since they are memory addresses. However, the data at the address stored in p is taken to be LONG (a 32-bit value), that at cptr is CHAR (an 8-bit value), that at i is INT (a 16-bit value), and that at gptr is gadget, which is an object (see OBJECT Type).

1.64 beginner.guide/Indirect types

Indirect types

In the previous example we saw INT and CHAR used as the destination types of pointers, and these are the 16- and 8-bit equivalents (respectively) of the LONG type. However, unlike LONG these types cannot be used directly to declare global or local variables, or procedure parameters. They can only be used in constructing types (for instance with PTR TO). The following declarations are therefore illegal, and it might be nice to try compiling a little program with such a declaration, just to see the error message the E compiler gives.

```
     /* This program fragment contains illegal declarations */
     DEF c:CHAR, i:INT
```

```
/* This program fragment contains illegal declarations */
PROC fred(a:INT, b:CHAR)
  DEF x:INT
  statements
ENDPROC
```

This is not much of a limitation because you can store INT or CHAR values in LONG variables if you really need to. However, it does mean there’s a nice, simple rule: every direct value in E is a 32-bit quantity, either a LONG or a pointer. In fact, LONG is actually short-hand for PTR TO CHAR, so you can use LONG values like they were actually PTR TO CHAR values.

1.65 beginner.guide/Finding addresses (making pointers)

Finding addresses (making pointers)

If a program knows the address of a variable it can directly read or alter the value stored in the variable. To obtain the address of a simple variable you use { and } around the variable name. The address of non-simple variables (e.g., objects and arrays) can be found much more easily (see the appropriate section), and in fact you will very rarely need to use {var }. However, if you understand how to explicitly make pointers with {var } and use the pointers to get to data, then you’ll understand the way pointers are used for the non-simple types much more quickly.

Addresses can be stored in a variable, passed to a procedure or whatever (they’re just 32-bit values). Try out the following program:

```
DEF x

PROC main()
  fred(2)
ENDPROC

PROC fred(y)
  DEF z
  WriteF('x is at address \d\n', {x})
  WriteF('y is at address \d\n', {y})
  WriteF('z is at address \d\n', {z})
  WriteF('fred is at address \d\n', {fred})
ENDPROC
```

Notice that you can also find the address of a procedure using { and }. This is is the memory location of the code the procedure represents. Here’s the output from one execution of this program (don’t expect your output to be exactly the same, though):

```
x is at address 3758280
y is at address 3758264
z is at address 3758252
fred is at address 3732878
```

This is an interesting program to run several times under different circumstances. You should see that sometimes the numbers for the addresses change. Running the program when another is multi-tasking (and eating memory) should produce the best changes, whereas running it consecutively (in one CLI) should produce the smallest (if any) changes. This gives you a glimpse at the complex memory handling of the Amiga and the E compiler.

1.66 beginner.guide/Extracting data (dereferencing pointers)

Extracting data (dereferencing pointers)

If you have an address stored in a variable (i.e., a pointer) you can extract the data using the ^ operator. This act of extracting data via a pointer is called dereferencing the pointer. This operator should only really be used when {var } has been used to obtain an address. To this end, LONG values are read and written when dereferencing pointers in this way. For pointers to non-simple types (e.g., objects and arrays), dereferencing is achieved in much more readable ways (see the appropriate section for details), and this operator is not used. In fact, ^var is seldom used in programs, but is useful for explaining how pointers work, especially in conjunction with {var }.

Using pointers can remove the scope restriction on local variables, i.e., they can be altered from outside the procedure for which they are local. Whilst this kind of use is not generally advised, it makes for a good example which shows the power of pointers. For example, the following program changes the value of the local variable x for the procedure fred from within the procedure barney.

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG
  x:=33
  p:={x}
  barney(p)
  WriteF('x is now \d\n', x)
ENDPROC

PROC barney(ptr:PTR TO LONG)
  DEF val
  val:=^ptr
  ^ptr:=val-6
ENDPROC
```

Here's what you can expect it to generate as output:

```
x is now 27
```

Notice that the ^ operator (i.e., dereferencing) is quite versatile. In the first assignment of the procedure barney it is used (with the pointer ptr) to get the value stored in the local variable x, and in the second it is used to change this variable's value. In either case, dereferencing makes the pointer behave exactly as if you'd written the variable for which it is a pointer. To emphasise this, we can remove the barney procedure, like we did above (see Style Reuse and Readability):

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG, val
  x:=33
  p:={x}
  val:=x
  x:=val-6
  WriteF('x is now \d\n', x)
ENDPROC
```

Everywhere the barney procedure used ^ptr we've written x (because we are now in the procedure for which x is local). We've also eliminated the ptr variable (the parameter to the barney procedure), since it was only used with the ^ operator.

To make things clear the fred and barney example is deliberately 'wordy'. The val and p variables are unnecessary, and the pointer types could be abbreviated to LONG or even omitted, for the reasons outlined above (see LONG Type). This is the compact form of the example:

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=33
  barney({x})
  WriteF('x is now \d\n', x)
ENDPROC

PROC barney(ptr)
  ^ptr:=^ptr-6
ENDPROC
```

By far the most common use of pointers is to address (or reference) large structures of data. It would be extremely expensive (in terms of CPU time) to pass large amounts of data from procedure to procedure, so addresses to such data are passed instead (and, as we know, these are just 32-bit values). The Amiga system functions (such as ones for creating windows) require a lot of structured data, so if you plan to do any real programming you are going to have to understand and use pointers.

1.67 beginner.guide/Procedure parameters

Procedure parameters

Only local and global variables have the luxury of a large choice of types. Procedure parameters can only be LONG or PTR TO type. This is not really a big limitation as we shall see in the later sections.

1.68 beginner.guide/ARRAY Type

ARRAY Type
==========

Quite often, the data used by a program needs to be ordered in some way, primarily so that it can be accessed easily. E provides a way to achieve such simple ordering: the ARRAY type. This type (in its various forms) is common to most computer languages.

Tables of data

Accessing array data

Array pointers

Point to other elements

Array procedure parameters

1.69 beginner.guide/Tables of data

Tables of data

Data can be grouped together in many different ways, but probably the most common and straight-forward way is to make a table. In a table the data is ordered either vertically or horizontally, but the important thing is the relative positioning of the elements. The E view of this kind of ordered data is the ARRAY type. An array is just a fixed sized collection of data in order. The size of an array is important and this is fixed when it is declared. The following illustrates array declarations:

```
DEF a[132]:ARRAY,
    table[21]:ARRAY OF LONG,
    ints[3]:ARRAY OF INT,
    objs[54]:ARRAY OF myobject
```

The size of the array is given in the square brackets ([and]). The type of the elements in the array defaults to CHAR, but this can be given explicitly using the OF keyword and the type name. However, only LONG, INT, CHAR and object types are allowed (LONG can hold pointer values so this isn't much of a limitation). Object types are described below (see
OBJECT Type
).

As mentioned above, procedure parameters cannot be arrays (see

Procedure parameters
). We will overcome this limitation soon (see

Array procedure parameters
).

1.70 beginner.guide/Accessing array data

Accessing array data

To access a particular element in an array you use square brackets again, this time specifying the index (or position) of the element you want. Indices start at zero for the first element of the array, one for the second element and, in general, (n-1) for the n-th element. This may seem strange at first, but it's the way most computer languages do it! We will see a reason why this makes sense soon (see
Array pointers
).

```
DEF a[10]:ARRAY

PROC main()
  DEF i
  FOR i:=0 TO 9
    a[i]:=i*i
  ENDFOR
  WriteF('The 7th element of the array a is \d\n', a[6])
  a[a[2]]:=10
  WriteF('The array is now:\n')
  FOR i:=0 TO 9
    WriteF(' a[\d] = \d\n', i, a[i])
  ENDFOR
ENDPROC
```

This should all seem very straight-forward although one of the lines looks a bit complicated. Try to work out what happens to the array after the assignment immediately following the first WriteF. In this assignment the index comes from a value stored in the array itself! Be careful when doing complicated things like this, though: make sure you don't try to read data from or write data to elements beyond the end of the array. In our example there are only ten elements in the array a, so it wouldn't be sensible to talk about the eleventh element. The program could have checked that the value stored at a[2] was a number between zero and nine before trying to access that array element, but it wasn't necessary in this case. Here's the output this example should generate:

```
The 7th element of the array a is 36
The array is now:
 a[0] = 0
 a[1] = 1
 a[2] = 4
 a[3] = 9
 a[4] = 10
 a[5] = 25
 a[6] = 36
 a[7] = 49
 a[8] = 64
 a[9] = 81
```

If you do try to write to a non-existent array element strange things can happen. This may be practically unnoticeable (like corrupting some other data), but if you're really unlucky you might crash your computer. The moral is: stay within the bounds of the array.

A short-hand for the first element of an array (i.e., the one with an index of zero) is to omit the index and write only the square brackets. Therefore, a[] is the same as a[0].

1.71 beginner.guide/Array pointers

```
                Array pointers
--------------
```

When you declare an array the address of the (beginning of the) array is given by the variable name without square brackets. Consider the following program:

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF ptr:PTR TO INT, i
  FOR i:=0 TO 9
    a[i]:=i
  ENDFOR
  ptr:=a
  ptr++
  ptr[]:=22
  FOR i:=0 TO 9
    WriteF('a[\d] is \d\n', i, a[i])
```

```
  ENDFOR
ENDPROC
```

Here’s the output from it:

```
a[0] is 0
a[1] is 22
a[2] is 2
a[3] is 3
a[4] is 4
a[5] is 5
a[6] is 6
a[7] is 7
a[8] is 8
a[9] is 9
```

You should notice that the second element of the array has been changed using the pointer. The ptr++ statement increments the pointer ptr to point to the next element of the array. It is important that ptr is declared as PTR TO INT since the array is an ARRAY OF INT. The [] is used to dereference the pointer and therefore 22 is stored in the second element of the array. In fact, the ptr can be used in exactly the same way as an array, so ptr[1] would be the next (or third element) of the array a (after the ptr++ statement). Also, since ptr points to the second element of a, negative values may legitimately be used as the index, and ptr[-1] is the first element of a.

In fact, the following declarations are identical except the first reserves an appropriate amount of memory for the array whereas the second relies on you having done this somewhere else in the program.

```
DEF a[20]:ARRAY OF INT

DEF a:PTR TO INT
```

The following diagram is similar to the diagrams given earlier (see

Addresses

). It is an illustration of an array, a, which was declared to be an array of twenty INTs.

```
         +--------+
         |Variable|
         |  ’a’   |
         |--------|
         | Address+----*
         +--------+     \
                         \
                          \
              +-------+ +--\----+ +-------+     +-------+ +-------+
              |Unknown| | a[0]  | | a[1]  |     | a[19] | |Unknown|
     Memory:  |+-----+| |+-----+| |+-----+| ... |+-----+| |+-----+|
              || XXX || || INT || || INT ||     || INT || || XXX ||
              +=======+ +=======+ +=======+     +=======+ +=======+
```

As you can see, the variable a is a pointer to the reserved chunk of memory which contains the array elements. Parts of memory that aren’t between a[0] and a[19] are marked as 'Unknown' because they are not part of the array. This memory should therefore not be accessed using the array a.

1.72 beginner.guide/Point to other elements

Point to other elements

We saw in the previous section how to increment a pointer so that it points to the next element in the array. Decrementing a pointer p (i.e., making it point to the previous element) is done in a similar way, using the p-- statement which works in the same way as the p++ statement. In fact, p++ and p-- are really expressions which denote pointer values. p++ denotes the address stored in p before it is incremented, and p-- denotes the address after p is decremented. Therefore,

```
       addr:=p
       p++
```

does the same as

```
       addr:=p++
```

And

```
       p--
       addr:=p
```

does the same as

```
       addr:=p--
```

The reason why ++ and -- should be used to increment and decrement a pointer is that values from different types occupy different numbers of memory locations. In fact, a single memory location is a byte, and this is eight bits. Therefore, CHAR values occupy a single byte, whereas LONG values take up four bytes (32 bits). If p were a pointer to CHAR and it was pointing to an array (of CHAR) the p+1 memory location would contain the second element of the array (and p+2 the third, etc.). But if p were a pointer to an array of LONG the second element in the array would be at p+4 (and the third at p+8). The locations p, p+1, p+2 and p+3 all make up the LONG value at address p. Having to remember things like this is a pain, and it's a lot less readable than using ++ or --. However, you must remember to declare your pointer with the correct type in order for ++ and -- to work correctly.

1.73 beginner.guide/Array procedure parameters

Array procedure parameters

Since we now know how to get the address of an array we can simulate passing an array as a procedure parameter by passing the address of the array. For example, the following program uses a procedure to fill in the first x elements of an array with their index numbers.

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF i
  fillin(a, 10)
  FOR i:=0 TO 9
    WriteF('a[\d] is \d\n', i, a[i])
  ENDFOR
ENDPROC

PROC fillin(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[]:=i
    ptr++
  ENDFOR
ENDPROC
```

Here's the output it should generate:

```
a[0] is 0
a[1] is 1
a[2] is 2
a[3] is 3
a[4] is 4
a[5] is 5
a[6] is 6
a[7] is 7
a[8] is 8
a[9] is 9
```

The array a only has ten elements so we shouldn't fill in any more than the first ten elements. Therefore, in the example, the call to the procedure fillin should not have a bigger number than ten as the second parameter. Also, we could treat ptr more like an array (and not use ++), but in this case using ++ is slightly better since we are assigning to each element in turn. The alternative definition of fillin (without using ++) is:

```
PROC fillin2(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[i]:=i
  ENDFOR
ENDPROC
```

Also, yet another version of fillin uses the expression form of ++ and the horizontal form of the FOR loop to give a really compact definition.

```
PROC fillin3(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1 DO ptr[]++:=i
ENDPROC
```

1.74 beginner.guide/OBJECT Type

OBJECT Type
===========

Objects are the E equivalent of C and Assembly structures, or Pascal records. They are like arrays except the elements are named not numbered, and the elements can be of different types. To find a particular element in an object you use a name instead of an index (number). Objects are also the basis of the OOP features of E (see Object Oriented E).

Example object

Element selection and element types

Amiga system objects

1.75 beginner.guide/Example object

Example object

We'll dive straight in with this first example, and define an object and use it. Object definitions are global and must be made before any procedure definitions.

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  DEF a:rec
  a.tag:=1
  a.check:=a
  a.data:=a.tag+(10000*a.tag)
ENDPROC
```

This program doesn't visibly do anything so there isn't much point in compiling it. What it does do, however, is show how a typical object is defined and elements of an object are selected.

The object being defined in the example is rec, and its elements are defined just like variable declarations (but without a DEF). There can be as many lines of element definitions as you like between the OBJECT and ENDOBJECT lines, and each line can contain any number of elements separated by commas. The elements of the rec object are tag and check (which are LONG), table (which is an array of CHAR with eight elements) and data (which is also LONG). Every variable of rec object type will have space reserved for each of these elements. The declaration of the (local) variable a therefore reserves enough memory for one rec object.

1.76 beginner.guide/Element selection and element types

Element selection and element types

To select elements in an object obj you use obj.name, where name is one of the element names. In the example, the tag element of the rec object a is selected by writing a.tag. The other elements are selected in a similar way.

Just like an array declaration the address of an object obj is stored in the variable obj, and any pointer of type PTR TO objectname can be used just like an object of type objectname. Therefore, in the previous example a is a PTR TO rec.

As the example object shows, the elements of an object can have several different types. In fact, the elements can have any type, including object, pointer to object and array of object. The following example shows how to access some different typed elements.

```
     OBJECT rec
       tag, check
       table[8]:ARRAY
       data:LONG
     ENDOBJECT

     OBJECT bigrec
       data:PTR TO LONG
       subrec:PTR TO rec
       rectable[22]:ARRAY OF rec
     ENDOBJECT

     PROC main()
       DEF r:rec, b:bigrec, rt:PTR TO rec
       r.table[]:="H"
       b.subrec:=r
       b.subrec.tag:=1
       b.subrec.data:=r.tag+(10000*b.subrec.tag)
       b.subrec.table[1]:="i"
       b.rectable[0].data:=r.tag
       b.rectable[0].table[0]:="A"
```

```
    rt:=b.rectable
    rt[].data++:=0
    rt[].table[]--:="B"
  ENDPROC
```

The ++ and -- operators apply to first thing in the selection (i.e., rt in both the last two assignments in the example above), and may only occur after all the selections. Notice that object selection and array indexing can be repeated as much as necessary (but only as the types of the elements allow). As a simple example, consider the third assignment:

```
    b.subrec.tag:=1
```

This selects the subrec element from the bigrec object b, and then sets the tag element of this rec object to 1. Now, consider one of the later assignments:

```
    b.rectable[0].table[0]:="A"
```

This selects the rectable element from b, which is an array of rec objects. The first element of this array is selected, and then the table element of the rec object is selected. Finally, the first character of the table is set to the ASCII value of character A.

As you can probably tell, it is important to give the elements of objects appropriate types if you want to do multiple selection in this way. However, this is not always possible or the best way of doing some things, so there is a way of giving a different type to pointers (this is called explicit pointer typing--see the ‘Reference Manual’ for more details).

Here’s a quite simple example which uses an array of objects:

```
  OBJECT rec
    tag, check
    table[8]:ARRAY
    data:LONG
  ENDOBJECT

  PROC main()
    DEF a[10]:ARRAY OF rec, p:PTR TO rec, i
    p:=a
    FOR i:=0 TO 9
      a[i].tag:=i
      p.check++:=i
    ENDFOR
    FOR i:=0 TO 9
      IF a[i].tag<>a[i].check
        WriteF('Whoops, a[\d] went wrong...\n', i)
      ENDIF
    ENDFOR
  ENDPROC
```

If you think about it for long enough you’ll see that a[0].tag is the same as a.tag. That’s because a is a pointer to the first element of the array, and the elements of the array are objects. Therefore, a is a pointer to an object (the first object in the array).

1.77 beginner.guide/Amiga system objects

Amiga system objects

There are many different Amiga system objects. For instance, there's one which contains the information needed to make a gadget (like the 'close' gadget on most windows), and one which contains all the information about a process or task. These objects are vitally important and so are supplied with E in the form of 'modules'. Each module is specific to a certain area of the Amiga system and contains object and other definitions. Modules are discussed in more detail later (see

Modules
).

1.78 beginner.guide/LIST and STRING Types

LIST and STRING Types
=====================

Arrays are common to many computer languages. However, they can be a bit of a pain because you always need to make sure you haven't run off the end of the array when you're writing to it. This is where the STRING and LIST types come in. STRING is very much like ARRAY OF CHAR and LIST is like ARRAY OF LONG. However, each has a set of E (built-in) functions which safely manipulate variables of these types without exceeding their bounds.

Normal strings and E-strings

String functions

Lists and E-lists

List functions

Complex types

Typed lists

Static data

1.79 beginner.guide/Normal strings and E-strings

Normal strings and E-strings

Normal strings are common to most programming languages. They are simply an array of characters, with the end of the string marked by a null character (ASCII zero). We’ve already met normal strings (see Strings).
The ones we used were constant strings contained in ’ characters, and they denote pointers to the memory where the string data is stored. Therefore, you can assign a string constant to a pointer (to CHAR), and you’ve got an array with ready-filled elements, i.e., an initialised array.

```
       DEF s:PTR TO CHAR
       s:='This is a string constant'
       /* Now s[] is T and s[2] is i */
```

Remember that LONG is actually PTR TO CHAR so this code is precisely the same as:

```
       DEF s
       s:='This is a string constant'
```

The following diagram illustrates the above assignment to s. The first two characters s[0] and s[1]) are T and h, and the last character (before the terminating null, or zero) is t. Memory marked as ‘Unknown’ is not part of the string constant.

```
        +--------+
        |Variable|
        |  ’s’   |
        |--------|
        |Address +----*
        +--------+     \
                        \
                         \
             +-------+ +--\----+ +-------+   +-------+ +-------+ +-------+
             |Unknown| | s[0]  | | s[1]  |   | s[24] | | s[25] | |Unknown|
     Memory: |+-----+| |+-----+| |+-----+|...|+-----+| |+-----+| |+-----+|
             || XXX || || "T" || || "h" ||   || "t" || ||  0  || || XXX ||
             +=======+ +=======+ +=======+   +=======+ +=======+ +=======+
```

E-strings are very similar to normal strings and, in fact, an E-string can be used wherever a normal string can. However, the reverse is not true, so if something requires an E-string you cannot use a normal string instead. The difference between a normal string and an E-string was hinted at in the introduction to this section: E-strings can be safely altered without exceeding their bounds. A normal string is just an array so you need to be careful not to exceed its bounds. However, an E-string knows what its bounds are, and so any of the string manipulation functions can alter them safely.

An E-string (STRING type) variable is declared as in the following example, with the maximum size of the E-string given just like an array declaration.

```
       DEF s[30]:STRING
```

As with an array declaration, the variable s is actually a pointer to the string data. To initialise an E-string you need to use the function StrCopy as we shall see.

There are some worked examples in Part Three (see String Handling and I-O) which show how to use normal strings and E-strings.

1.80 beginner.guide/String functions

String functions

There are a number of useful built-in functions which manipulate strings. Remember that if an E-string can be used wherever a normal string can, but normal strings cannot be used where an E-string is required. If a parameter is marked as string then a normal or E-string can be passed as that parameter, but if it is marked as e-string then only an E-string may be used. Some of these functions have default arguments, which means you don't need to specify some parameters to get the default values (see Default Arguments). (You can, of course, ignore the defaults and always give all parameters.)

String(maxsize)

Allocates memory for an E-string of maximum size maxsize and returns a pointer to the string data. It is used to make space for a new E-string, like a STRING declaration does. The following code fragments are practically equivalent:

```
DEF s[37]:STRING

DEF s:PTR TO CHAR
s:=String(37)
```

The slight difference is that there may not be enough memory left to hold the E-string when the String function is used. In that case the special value NIL (a constant) is returned. Your program must check that the value returned is not NIL before you use it as an E-string (or dereference it). The memory for the declaration version is allocated when the program is run, so your program won't run if there isn't enough memory. The String version is often called dynamic allocation because it happens only when the program is running; the declaration version has allocation done by the E compiler. The memory allocated using String is deallocated using DisposeLink (see

System support functions
).

StrCmp(string1,string2,length=ALL)

Compares string1 with string2 (they can both be normal or E-strings). Returns TRUE if the first length characters of the strings match, and FALSE otherwise. The length defaults to the special constant ALL which means that the strings must agree on every character. For example, the following comparisons all return TRUE:

```
StrCmp('ABC',  'ABC')
StrCmp('ABC',  'ABC', ALL)
StrCmp('ABCd', 'ABC',    3)
StrCmp('ABCde','ABCxxjs',3)
```

And the following return FALSE (notice the case of the letters):

```
StrCmp('ABC',  'ABc')
StrCmp('ABC',  'ABc', ALL)
StrCmp('ABCd', 'ABC', ALL)
```

StrCopy(e-string,string,length=ALL)

Copies the contents of string to e-string, and also returns a pointer to the resulting E-string (for convenience). Only length characters are copied from the source string, but the special constant ALL can be used to indicate that the whole of the source string is to be copied (and this is the default value for length). Remember that E-strings are safely manipulated, so the following code fragment results in s becoming More th, since its maximum size is (from its declaration) seven characters.

```
DEF s[7]:STRING
StrCopy(s, 'More than seven characters', ALL)
```

A declaration using STRING (or ARRAY) reserves a small part of memory, and stores a pointer to this memory in the variable being declared. So to get data into this memory you need to copy it there, using StrCopy. If you're familiar with very high-level languages like BASIC you should take care, because you might think you can assign a string to an array or an E-string variable. In E (and languages like C and Assembly) you must explicitly copy data into arrays and E-strings. You should not do the following:

```
/* You don't want to do things like this! */
DEF s[80]:STRING
s:='This is a string constant'
```

This is fairly disastrous: it throws away the pointer to reserved memory that was stored in s and replaces it by a pointer to the string constant. s is then no longer an E-string, and cannot be repaired using SetStr. If you want s to contain the above string you must use StrCopy:

```
DEF s[80]:STRING
StrCopy(s,'This is a string constant')
```

The moral is: remember when you are using pointers to data and when you need to copy data. Also, remember that assignment does not copy large arrays of data, it only copies pointers to data, so if you want to store some data in an ARRAY or STRING type variable you need to copy it there.

StrAdd(e-string,string,length=ALL)

This does the same as StrCopy but the source string is copied onto the end of the destination E-string. The following code fragment results in s becoming This is a string and a half.

```
DEF s[30]:STRING
StrCopy(s, 'This is a string', ALL)
StrAdd(s,  ' and a half')
```

StrLen(string)

Returns the length of string. This assumes that the string is terminated by a null character (i.e., ASCII zero), which is true for any strings made from E-strings and string constants. However, you can make a string constant look short if you use the null character (the special sequence \0) in it. For instance, these calls all return three:

```
StrLen('abc')
StrLen('abc\0def')
```

In fact, most of the string functions assume strings are null-terminated, so you shouldn't use null characters in your strings unless you really know what you're doing.

For E-strings StrLen is less efficient than the EstrLen function.

EstrLen(e-string)

Returns the length of e-string (remember this can only be an E-string). This is much more efficient than StrLen since E-strings know their length and it doesn't need to search the string for a null character.

StrMax(e-string)

Returns the maximum length of e-string. This is not necessarily the current length of the E-string, rather it is the size used in the declaration with STRING or the call to String.

RightStr(e-string1,e-string2,length)

This is like StrCopy but it copies the right-most characters from e-string2 to e-string1 and both strings must be E-strings. At most length characters are copied, and the special constant ALL cannot be used (to copy all the string you should, of course, use StrCopy). For instance, a value of one for length means the last character of e-string2 is copied to e-string1.

MidStr(e-string,string,index,length=ALL)

Copies the contents of string starting at index (which is an index just like an array index) to e-string. At most length characters are copied, and the special constant ALL can be used if all the remaining characters in string should be copied (this is the default value for length). For example, the following two calls to MidStr result in s becoming four:

```
DEF s[30]:STRING
MidStr(s, 'Just four',      5)
MidStr(s, 'Just four apples', 5, 4)
```

InStr(string1,string2,startindex=0)

Returns the index of the first occurrence of string2 in string1 starting at startindex (in string1). startindex defaults to zero. If string2 could not be found then -1 is returned.

TrimStr(string)

Returns the address of (i.e., a pointer to) the first non-whitespace character in string. For instance, the following code fragment results in s becoming 12345.

```
DEF s:PTR TO CHAR
s:=TrimStr('  \n \t   12345')
```

LowerStr(string)

Converts all uppercase letters in string to lowercase. This change is made in-place, i.e., the contents of the string are directly affected. The string is returned for convenience.

UpperStr(string)

Converts all lowercase letters in string to uppercase. Again, this change is made in-place and the string is returned for convenience.

SetStr(e-string,length)

Sets the length of e-string to length. E-strings know how long they are, so if you alter an E-string (without using an E-string function) and change its size you need to set its length using this function before you can use it as an E-string again. For instance, if you've used an E-string like an array (which you can do) and written characters to it directly you must set its length before you can treat it as anything other than an array/string:

```
DEF s[10]:STRING
s[0]:="a"     /* Remember that "a" is a character value. */
s[1]:="b"
s[2]:="c"
s[3]:="d"     /* At this point s is just an array of CHAR. */
SetStr(s, 4)  /* Now, s can be used as an E-string again.  */
SetStr(s, 2)  /* s is a bit shorter, but still an E-string.*/
```

Notice that this function can be used to shorten an E-string (but you cannot lengthen it this way).

Val(string,address=NIL)

What this function does is straight-forward but how you use it is a bit complicated. Basically, it converts string to a LONG integer. Leading whitespace is ignored, and a leading % or $ means that the string denotes a binary or hexadecimal integer (in the same way they do for numeric constants). The decoded integer is returned as the regular return value (see Multiple Return Values). The number of characters of string that were read to make the integer is stored at address, which is usually a variable address (from using {var }), and is returned as the first optional return value. If address is the special constant NIL (or zero) then this number is not stored (this is the default value for address). You can use this number to calculate the position in the string which was not part of the integer in the string. If an integer could not be decoded from the string then zero is returned and zero is stored at address.

Follow the comments in this example, and pay special attention to the use of the pointer p.

```
DEF s[30]:STRING, value, chars, p:PTR TO CHAR
StrCopy(s, ' \t \n 10 \t $3F -%0101010')
value, chars:=Val('abcde 10 20')  -> Two return values...
  /* After the above line, value and chars will both be zero */
value:=Val(s, {chars})          -> Use address of chars
  /* Now value will be 10, chars will be 7 */
p:=s+chars
  /* p now points to the space after the 10 in s */
value, chars:=Val(p)
  /* Now value will be $3F (63), chars will be 6 */
p:=p+chars
  /* p now points to the space after the $3F in s */
value, chars:=Val(p)
  /* Now value will be -%0101010 (-42), chars will be 10 */
```

Notice the two different ways of finding the number of characters read: a multiple-assignment and using the address of a variable.

There's a couple of other string functions (ReadStr and StringF) which will be discussed later (see Input and output functions).

1.81 beginner.guide/Lists and E-lists

Lists and E-lists

Lists are just like strings with LONG elements rather than CHAR elements (so they are very much like ARRAY OF LONG). The list equivalent of an E-string is something called an E-list. It has the same properties as an E-string, except the elements are LONG (so could be pointers). Normal lists are most like string constants, except that the elements can be built from variables and so do not have to be constants. Just as strings are not true E-strings, (normal) lists are not true E-lists.

Lists are written using [and] to delimit comma separated elements. Like string constants a list returns the address of the memory which contains the elements.

For example the following code fragment:

```
DEF list:PTR TO LONG, number
number:=22
list:=[1,2,3,number]
```

is equivalent to:

```
DEF list[4]:ARRAY OF LONG, number
number:=22
list[0]:=1
list[1]:=2
list[2]:=3
list[3]:=number
```

Now, which of these two versions would you rather write? As you can see, lists are pretty useful for making your program easier to write and much easier to read.

E-list variables are like E-string variables and are declared in much the same way. The following code fragment declares lt to be an E-list of maximum size 30. As ever, lt is then a pointer (to LONG), and it points to the memory allocated by the declaration.

```
DEF lt[30]:LIST
```

Lists are most useful for writing tag lists, which are increasingly used in important Amiga system functions. A tag list is a list where the elements are thought of in pairs. The first element of a pair is the tag, and the second is some data for that tag. See the ‘Rom Kernel Reference Manual (Libraries)’ for more details.

1.82 beginner.guide/List functions

List functions

There are a number of list functions which are very similar to the string functions (see String functions). Remember that E-lists are the list equivalents of E-strings, i.e., they can be altered and extended safely without exceeding their bounds. As with E-strings, E-lists are downwardly compatible with lists. Therefore, if a function requires a list as a parameter you can supply a list or an E-list. But if a function requires an E-list you cannot use a list in its place.

List(maxsize)
Allocates memory for an E-list of maximum size maxsize and returns a pointer to the list data. It is used to make space for a new E-list, like a LIST declaration does. The following code fragments are (as with String) practically equivalent:

```
DEF lt[46]:LIST
```

```
DEF lt:PTR TO LONG
lt:=List(46)
```

Remember that you need to check that the return value from List is not NIL before you use it as an E-list. Like String, the memory allocated using List is deallocated using DisposeLink (see

System support functions
).

ListCmp(list1,list2,length=ALL)

Compares list1 with list2 (they can both be normal or E-lists). Works just like StrCmp does for E-strings, so, for example, the following comparisons all return TRUE:

```
ListCmp([1,2,3,4],   [1,2,3,4])
ListCmp([1,2,3,4],   [1,2,3,7], 3)
ListCmp([1,2,3,4,5], [1,2,3],   3)
```

ListCopy(e-list,list,length=ALL)

Works just like StrCopy, and the following example shows how to initialise an E-list:

```
DEF lt[7]:LIST, x
x:=4
ListCopy(lt, [1,2,3,x])
```

As with StrCopy, an E-list cannot be over-filled using ListCopy.

ListAdd(e-list,list,length=ALL)

Works just like StrAdd, so this next code fragment results in the E-list lt becoming the E-list version of [1,2,3,4,5,6,7,8].

```
DEF lt[30]:LIST
ListCopy(lt, [1,2,3,4])
ListAdd(lt, [5,6,7,8])
```

ListLen(list)

Works just like StrLen, returning the length of list. There is no E-list specific length function.

ListMax(e-list)

Works just like StrMax, returning the maximum length of the e-list.

SetList(e-list,length)

Works just like SetStr, setting the length of e-list to length.

ListItem(list,index)

Returns the element of list at index. For example, if lt is an E-list then ListItem(lt,n) is the same as lt[n]. This function is most useful when the list is not an E-list. For example, the following two code fragments are equivalent:

```
WriteF(ListItem(['Fred','Barney','Wilma','Betty'], name))
```

```
DEF lt:PTR TO LONG
```

```
            lt:=['Fred','Barney','Wilma','Betty']
            WriteF(lt[name])
```

1.83 beginner.guide/Complex types

Complex types

In E the STRING and LIST types are called complex types. Complex typed variables can also be created using the String and List functions as we've seen in the previous sections.

1.84 beginner.guide/Typed lists

Typed lists

Normal lists contain LONG elements, so you can write initialised arrays of LONG elements. What about other kinds of array? Well, that's what typed lists are for. You specify the type of the elements of a list using :type after the closing]. The allowable types are CHAR, INT, LONG and any object type. There is a subtle difference between a normal, LONG list and a typed list (even a LONG typed list): only normal lists can be used with the list functions (see
List functions
). For this reason,
the term 'list' tends to refer only to normal lists.

The following code fragment uses the object rec defined earlier (see

Example object
) and gives a couple of examples of typed lists:

```
       DEF ints:PTR TO INT, objects:PTR TO rec, p:PTR TO CHAR
       ints:=[1,2,3,4]:INT
       p:='fred'
       objects:=[1,2,p,4,
                 300,301,'barney',303]:rec
```

It is equivalent to:

```
       DEF ints[4]:ARRAY OF INT, objects[2]:ARRAY OF rec, p:PTR TO CHAR
       ints[0]:=1
       ints[1]:=2
       ints[2]:=3
       ints[3]:=4
       p:='fred'
       objects[0].tag:=1
       objects[0].check:=2
```

```
objects[0].table:=p
objects[0].data:=4
objects[1].table:='barney'
objects[1].tag:=300
objects[1].data:=303
objects[1].check:=301
```

The last group of assignments to objects[1] have deliberately been shuffled in order to emphasise that the order of the elements in the definition of the object rec is significant. Each of the elements of the list corresponds to an element in the object, and the order of elements in the list corresponds to the order in the object definition. In the example, the (object) list assignment line was broken after the end of the first object (the fourth element) to make it a bit more readable. The last object in the list need not be completely defined, so, for instance, the second line of the assignment could have contained only three elements. This makes an object-typed list slightly different from the corresponding array of objects, since an array always defines a whole number of objects. With an object-typed list you must be careful not to access the undefined elements of a partially defined trailing object.

1.85 beginner.guide/Static data

Static data

String constants (e.g., fred), lists (e.g., [1,2,3]) and typed lists (e.g., [1,2,3]:INT) are static data. This means that the address of the (initialised) data is fixed when the program is run. Normally you don't need to worry about this, but, for instance, if you want to have a series of lists as initialised arrays you might be tempted to use some kind of loop:

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=[1, i, i*i]
      /* This assignment is probably not what you want! */
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    WriteF('a[\d] is an array at address \d\n', i, p)
    WriteF('  and the second element is \d\n', p[1])
  ENDFOR
ENDPROC
```

The array a is an array of pointers to initialised arrays (which are all three elements long). But, as the comment suggests and the program shows, this probably doesn't do what was intended, since the list is static. That means the address of the list is fixed, so each element of a gets the same address (i.e., the same array). Since i is used in the list the contents of that part of memory varies slightly as the first FOR loop is processed. But after this loop the contents remain fixed, and the second element of each of the ten arrays is always nine. This is an example of the output that will be generated (the ... represents a number of similar lines):

```
a[0] is an array at address 4021144
  and the second element is 9
a[1] is an array at address 4021144
  and the second element is 9
...
a[9] is an array at address 4021144
  and the second element is 9
```

One solution is to use the dynamic typed-allocation operator NEW (see

NEW and END Operators

). Another solution is to use the function List and copy the normal list into the new E-list using ListCopy:

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=List(3)
    /* Must check that the allocation succeeded before copying */
    IF a[i]<>NIL THEN ListCopy(a[i], [1, i, i*i], ALL)
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    IF p=NIL
      WriteF('Could not allocate memory for a[\d]\n', i)
    ELSE
      WriteF('a[\d] is an array at address \d\n', i, p)
      WriteF('  and the second element is \d\n', p[1])
    ENDIF
  ENDFOR
ENDPROC
```

The problem is not so bad with string constants, since the contents are fixed. However, if you alter the contents explicitly, you will need to take care not to run into the same problem, as this next example shows.

```
PROC main()
  DEF i, strings[10]:ARRAY OF LONG, s:PTR TO CHAR
  FOR i:=0 TO 9
    strings[i]:='Hello World\n'
      /* This assignment is probably not what you want! */
  ENDFOR
  s:=strings[4]
  s[5]:="X"
  FOR i:=0 TO 9
    WriteF('strings[\d] is ', i)
    WriteF(strings[i])
  ENDFOR
ENDPROC
```

This is an example of the output that will be generated (again, the ... represents a number of similar lines)::

```
strings[0] is HelloXWorld
strings[1] is HelloXWorld
...
strings[9] is HelloXWorld
```

Again, the solution is to use dynamic allocation. The functions String and StrCopy should be used in the same way that List and ListCopy were used above.

1.86 beginner.guide/Linked Lists

Linked Lists
============

E-lists and E-strings have a useful extension: they can be used to make linked lists. These are like the lists we've seen already, except the list elements do not occupy a contiguous block of memory. Instead, each element has an extra piece of data: a pointer to the next element in the list. This means that each element can be anywhere in memory. (Normally, the next element of a list is in the next position in memory.) The end of a linked list has been reached when the pointer to the next element is the special value NIL (a constant). You need to be very careful to check that the pointer is not NIL because if you dereference a NIL pointer the program will most definitely crash.

The elements of a linked list are E-lists or E-strings (i.e., the elements are complex typed). So, you can link E-lists to get a 'linked list of E-lists' (or, more simply, a 'list of lists'). Similarly, linking E-strings gives 'linked list of E-strings', or a 'list of strings'. You don't have to stick to these two kinds of linked lists, though: you can use a mixture of E-lists and E-strings in the same linked list. To link one complex typed element to another you use the Link function and to find subsequent elements in a linked list you use the Next or Forward functions.

Link(complex1,complex2)

Links complex1 to complex2. Both must be an E-list or an E-string, with the exception that complex2 can be the special constant NIL to indicate that complex1 is the end of the linked list. The value complex1 is returned by the function, which isn't always useful so, usually, calls to Link will be used as statements rather than functions. The effect of Link is that complex1 will point to complex2 as the next element in the linked list (so complex1 is the head of the list, and complex2 is the tail). For both E-lists and E-strings the pointer to the next element is initially NIL, so you will only need to use Link with a NIL parameter when you want to make a linked list shorter (by losing the tail).

Next(complex)

Returns the pointer to the next element in the linked list. This may be the special constant NIL if complex is the last element in the linked list. Be careful to check that the value isn't NIL before you dereference it! Follow the comments in the example below:

```
DEF s[23]:STRING, t[7]:STRING, lt[41]:LIST, lnk
/* The next two lines set up the linked list "lnk" */
lnk:=Link(lt,t)  /* lnk list starts at lt and is lt->t    */
lnk:=Link(s,lt)  /*   Now it starts at s  and is s->lt->t */
/* The next three lines follow the links in "lnk"  */
lnk:=Next(lnk)   /*   Now it starts at lt and is lt->t    */
lnk:=Next(lnk)   /*   Now it starts at t  and is t        */
lnk:=Next(lnk)   /* Now lnk is NIL so the list has ended  */
```

You may safely call Next with a NIL parameter, and in this case it will return NIL.

Forward(complex,expression)

Returns a pointer to the element which is expression number of links down the linked list complex. If expression represents the value one a pointer to the next element is returned (just like using Next). If it's two a pointer to the element after that is returned.

If expression represents a number which is greater than the number of links in the list the special value NIL is returned.

Since the link in a linked list is a pointer to the next element you can only look through the list from beginning to end. Technically this is a singly linked list (a doubly linked list would also have a pointer to the previous element in the list, enabling backwards searching through the list).

Linked lists are useful for building lists that can grow quite large. This is because it's much better to have lots of small bits of memory than a large lump. However, you only need to worry about these things when you're playing with quite big lists (as a rough guide, ones with over 100,000 elements are big!).

1.87 beginner.guide/More About Statements and Expressions

More About Statements and Expressions

****

This chapter details various E statements and expressions that were not covered in Part One. It also completes some of the partial descriptions given in Part One.

Turning an Expression into a Statement

Initialised Declarations

Assignments

More Expressions

More Statements

Unification

Quoted Expressions

Assembly Statements

1.88 beginner.guide/Turning an Expression into a Statement

Turning an Expression into a Statement
==

The VOID operator converts an expression to a statement. It does this by evaluating the expression and then throwing the result away. This may not seem very useful, but in fact we've done it a lot already. We didn't use VOID explicitly because E does this automatically if it finds an expression where it was expecting a statement (normally when it is on a line by itself). Some of the expressions we've turned into statements were the procedure calls (to WriteF and fred) and the use of ++. Remember that all procedure calls denote values because they're really functions that, by default, return zero (see Functions).

For example, the following code fragments are equivalent:

```
VOID WriteF('Hello world\n')
VOID x++

WriteF('Hello world\n')
x++
```

Since E automatically uses VOID it's a bit of a waste of time writing it in, although there may be occasions where you want to use it to make this voiding process more explicit (to the reader). The important thing is the fact that expressions can validly be used as statements in E.

1.89 beginner.guide/Initialised Declarations

Initialised Declarations
========================

Some variables can be initialised using constants in their declarations. The variables you cannot initialise in this way are array and complex type variables (and procedure parameters, obviously). All the other kinds can be initialised, whether they are local or global. An initialised declaration looks very much like a constant definition, with the value following the variable name and a = character joining them. The following example illustrates initialised declarations:

```
     SET ENGLISH, FRENCH, GERMAN, JAPANESE, RUSSIAN

     CONST FREDLANGS=ENGLISH OR FRENCH OR GERMAN

     DEF fredspeak=FREDLANGS,
         p=NIL:PTR TO LONG, q=0:PTR TO rec

     PROC fred()
       DEF x=1, y=88
       /* Rest of procedure */
     ENDPROC
```

Notice how the constant FREDLANGS needs to be defined in order to initialise the declaration of fredspeak to something mildly complicated. Also, notice the initialisation of the pointers p and q, and the position of the type information.

Of course, if you want to initialise variables with anything more complicated than a constant you can use assignments at the start of the code. Generally, you should always initialise your variables (using either method) so that they are guaranteed to have a sensible value when you use them. Using the value of a variable that you haven't initialised in some way will probably get you in to a lot of trouble, because the value will just be some random value that happened to be in the memory used by the variable. There are rules for how E initialises some kinds of variables (see the ‘Reference Manual’, but it's wise to explicitly initialise even those, as (strangely enough!) this will make your program more readable.

1.90 beginner.guide/Assignments

Assignments
===========

We've already seen some assignments--these were assignment statements. Assignment expressions are similar except (as you've guessed) they can be used in expressions. This is because they return the value on the right-hand side of the assignment as well as performing the assignment. This is useful for efficiently checking that the value that's been assigned is sensible. For instance, the following code fragments are equivalent, but the first uses an assignment expression instead of a normal assignment statement.

```
       IF (x:=y*z)=0
         WriteF('Error: y*z is zero (and x is zero)\n')
       ELSE
         WriteF('OK: y*z is not zero (and x is y*z)\n')
       ENDIF

       x:=y*z
       IF x=0
         WriteF('Error: y*z is zero (and x is zero)\n')
       ELSE
         WriteF('OK: y*z is not zero (and x is y*z)\n')
```

```
       ENDIF
```

You can easily tell the assignment expression: it's in parentheses and not on a line by itself. Notice the use of parentheses to group the assignment expression. Technically, the assignment operator has a very low precedence. Less technically, it will take as much as it can of the right-hand side to form the value to be assigned, so you need to use parentheses to stop x getting the value ((y*z)=0) (which will be TRUE or FALSE, i.e., -1 or zero).

Assignment expressions, however, don't allow as rich a left-hand side as assignment statements. The only thing allowed on the left-hand side of an assignment expression is a variable name, whereas the statement form allows:

```
       var
       var [ expression ]
       var . obj_element_name
       ^ var
```

(With as many repetitions of object element selection and/or array indexing as the elements' types allow.) Each of these may end with ++ or --. Therefore, the following are all valid assignments (the last three use assignment expressions):

```
       x:=2
       x--:=1
       x[a*b]:=rubble
       x.apple++:=3
       x[22].apple:=y*z
       x[].banana.basket[6]:=3+full(9)
       x[].pear--:=fred(2,4)

       x.pear:=(y:=2)
       x[y*z].table[1].orange:=(IF (y:=z)=2 THEN 77 ELSE 33)
       WriteF('x is now \d\n', x:=1+(y:=(z:=fred(3,5)/2)*8))
```

You may be wondering what the ++ or -- affect. Well, it's very simple: they only affect the var, which is x in all of the examples above. Notice that x[].pear-- is the same as x.pear--, for the same reasons mentioned earlier (see
Element selection and element types
).

1.91 beginner.guide/More Expressions

More Expressions
================

This section discusses side-effects, details two new operators (BUT and SIZEOF) and completes the description of the AND and OR operators.

Side-effects

BUT expression

Bitwise AND and OR

SIZEOF expression

1.92 beginner.guide/Side-effects

Side-effects

If evaluating an expression causes the contents of variables to change then that expression is said to have side-effects. An assignment expression is a simple example of an expression with side-effects. Less obvious ones involve function calls with pointers to variables. Generally, expressions with side-effects should be avoided unless it is really obvious what is happening. This is because it can be difficult to find problems with your program's code if subtleties are buried in complicated expressions. On the other hand, side-effecting expressions are concise and often very elegant. They are also useful for obfuscating your code (i.e., making it difficult to understand--a form of copy protection!).

1.93 beginner.guide/BUT expression

BUT expression

BUT is used to sequence two expressions. exp1 BUT exp2 evaluates exp1, and then evaluates and returns the value of exp2. This may not seem very useful at first sight, but if the first expression is an assignment it allows for a more general assignment expression. For example, the following code fragments are equivalent:

```
       fred((x:=12*3) BUT x+y)

       x:=12*3
       fred(x+y)
```

Notice that parentheses need to be used around the assignment expression (in the first fragment) for the reasons given earlier (see Assignments).

1.94 beginner.guide/Bitwise AND and OR

Bitwise AND and OR

As hinted in the earlier chapters, the operators AND and OR are not simply logical operators. In fact, they are both bit-wise operators, where a bit is a binary digit (i.e., the zeroes or ones in the binary form of a number). So, to see how they work we should look at what happens to zeroes and ones:

x	y	x OR y	x AND y
1	1	1	1
1	0	1	0
0	1	1	0
0	0	0	0

Now, when you AND or OR two numbers the corresponding bits (binary digits) of the numbers are compared individually, according to the above table. So if x were %0111010 and y were %1010010 then x AND y would be %0010010 and x OR y would be %1111010:

```
 %0111010        %0111010
AND             OR
 %1010010        %1010010
  -------         -------
 %0010010        %1111010
```

The numbers (in binary form) are lined up above each other, just like you do additions with normal numbers (i.e., starting with the right-hand digits, and maybe padding with zeroes on the left-hand side). The two bits in each column are AND-ed or OR-ed to give the result below the line.

So, how does this work for TRUE and FALSE and logic operations? Well, FALSE is the number zero, so all the bits of FALSE are zeroes, and TRUE is -1, which is has all 32 bits as ones (these numbers are LONG so they are 32-bit quantities). So AND-ing and OR-ing these values always gives numbers which have all zero bits (i.e., FALSE) or all one bits (i.e., TRUE), as appropriate. It’s only when you start mixing numbers that aren’t zero or -1 that you can muck up the logic. The non-zero numbers one and four are (by themselves) considered to be TRUE, but 4 AND 1 is %100 AND %001 which is zero (i.e., FALSE). So when you use AND as the logical operator it’s not strictly true that all non-zero numbers represent TRUE. OR does not give such problems so all non-zero numbers are treated as TRUE. Run this example to see why you should be careful:

```
PROC main()
  test(TRUE,          ’TRUE\t\t’)
  test(FALSE,         ’FALSE\t\t’)
  test(1,             ’1\t\t’)
  test(4,             ’4\t\t’)
  test(TRUE OR TRUE,  ’TRUE OR TRUE\t’)
  test(TRUE AND TRUE, ’TRUE AND TRUE\t’)
  test(1 OR 4,        ’1 OR 4\t\t’)
  test(1 AND 4,       ’1 AND 4\t\t’)
```

```
ENDPROC

PROC test(x, title)
  WriteF(title)
  WriteF(IF x THEN ' is TRUE\n' ELSE ' is FALSE\n')
ENDPROC
```

Here's the output that should be generated:

```
TRUE             is TRUE
FALSE            is FALSE
1                is TRUE
4                is TRUE
TRUE OR TRUE     is TRUE
TRUE AND TRUE    is TRUE
1 OR 4           is TRUE
1 AND 4          is FALSE
```

So, AND and OR are primarily bit-wise operators, and they can be used as logical operators under most circumstances, with zero representing false and all other numbers representing true. Care must be taken when using AND with some pairs of non-zero numbers, since the bit-wise AND of such numbers does not always give a non-zero (or true) result.

1.95 beginner.guide/SIZEOF expression

SIZEOF expression

SIZEOF returns the size, in bytes, of an object or a built-in type (like LONG). This can be useful for determining storage requirements. For instance, the following code fragment prints the size of the object rec:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  WriteF('Size of rec object is \d bytes\n', SIZEOF rec)
ENDPROC
```

You may think that SIZEOF is unnecessary because you can easily calculate the size of an object just by looking at the sizes of the elements. Whilst this is generally true (it was for the rec object), there is one thing to be careful about: alignment. This means that ARRAY, INT, LONG and object typed elements must start at an even memory address. Normally this isn't a problem, but if you have an odd number of consecutive CHAR typed elements or an odd sized ARRAY OF CHAR, an extra, pad byte is introduced into the object so that the following element is aligned properly. This pad byte can be considered part of an ARRAY OF CHAR, so in effect this means array sizes are rounded up to the nearest even number. Otherwise, pad bytes are just an unusable part of an object, and their presence means the object size is not quite what you'd expect. Try the following program:

```
OBJECT rec2
  tag, check
  table[7]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  WriteF('Size of rec2 object is \d bytes\n', SIZEOF rec2)
ENDPROC
```

The only difference between the rec and rec2 objects is that the array size is seven in rec2. If you run the program you'll see that the size of the object has not changed. We might just as well have declared the table element to be a slightly bigger array (i.e., have eight elements).

1.96 beginner.guide/More Statements

More Statements
===============

This section details five new statements (INC, DEC, JUMP, EXIT and LOOP) and describes the use of labelling.

INC and DEC statements

Labelling and the JUMP statement

EXIT statement

LOOP block

1.97 beginner.guide/INC and DEC statements

INC and DEC statements

INC x is the same as the statement x:=x+1. However, because it doesn't do an addition it's a bit more efficient. Similarly, DEC x is the same as x:=x-1.

The observant reader may think that INC and DEC are the same as ++ and --. But there's one important difference: INC x always increases x by one, whereas x++ may increase x by more than one depending on the type to which x points. For example, if x were a pointer to INT then x++ would increase x by two (INT is 16-bit, which is two bytes).

1.98 beginner.guide/Labelling and the JUMP statement

```
                Labelling and the JUMP statement
--------------------------------
```

A label names a position in a program, and these names are global (they can be used in any procedure). The most common use of label is with the JUMP statement, but you can also use labels to mark the position of some data (see
Assembly Statements
). To define a label you write a name followed by a colon immediately before the position you want to mark. This must be just before the beginning of a statement, either on the previous line (by itself) or the start of the same line.

The JUMP statement makes execution continue from the position marked by a label. This position must be in the same procedure, but it can be, for instance, outside of a loop (and the JUMP will then have terminated that loop). For example, the following code fragments are equivalent:

```
       x:=1
       y:=2
       JUMP rubble
       x:=9999
       y:=1234
     rubble:
       z:=88

       x:=1
       y:=2
       z:=88
```

As you can see the JUMP statement has caused the second group of assignments to x and y to be skipped. A more useful example uses JUMP to help terminate a loop:

```
       x:=1
       y:=2
       WHILE x<10
         IF y<10
           WriteF('x is \d, y is \d\n', x, y)
         ELSE
           JUMP end
         ENDIF
         x:=x+2
         y:=y+2
       ENDWHILE
     end:
       WriteF('Finished!\n')
```

This loop terminates if x is not less than ten (the WHILE check), or if y is not less than ten (the JUMP in the IF block). This may seem pretty familiar, because it's practically the same as an example earlier (see

WHILE loop
). In fact, it's equivalent to:

```
x:=1
y:=2
WHILE (x<10) AND (y<10)
  WriteF('x is \d, y is \d\n', x, y)
  x:=x+2
  y:=y+2
ENDWHILE
WriteF('Finished!\n')
```

1.99 beginner.guide/EXIT statement

EXIT statement

As noted above, you can use the JUMP statement and labelling to break out of a loop prematurely. However, a much nicer mechanism exists for WHILE and FOR loops: the EXIT statement. This statement will terminate the closest one of these loops (of which it is part) if the supplied expression evaluates to true (i.e., a non-zero value). Any loop using EXIT can be re-written without it, but sometimes at the expense of readability.

The following fragments of code are equivalent:

```
  FOR x:=1 TO 10
    y:=f(x)
  EXIT y=-1
    WriteF('x=\d, f(x)=\d\n', x, y)
  ENDFOR

  FOR x:=1 TO 10
    y:=f(x)
    IF y=-1 THEN JUMP end
    WriteF('x=\d, f(x)=\d\n', x, y)
  ENDFOR
end:
```

This example shows a situation which is arguably more readable using something like EXIT. It can be rewritten using a WHILE loop, as below, but the code is a bit less clear.

```
going:=TRUE
x:=1
WHILE going AND (x<=10)
  y:=f(x)
  IF y=-1
    going:=FALSE
  ELSE
```

```
          WriteF('x=\d, f(x)=\d\n', x, y)
          INC x
        ENDIF
      ENDWHILE
```

1.100 beginner.guide/LOOP block

LOOP block

A LOOP block is a multi-line statement. It's the general form of loops like the WHILE loop, and it builds a loop with no check. So, this kind of loop would normally never end. However, as we now know, you can terminate a LOOP block using the JUMP statement. As an example, the following two code fragments are equivalent:

```
       x:=0
       LOOP
         IF x<100
           WriteF('x is \d\n', x++)
         ELSE
           JUMP end
         ENDIF
       ENDLOOP
     end:
       WriteF('Finished\n')

       x:=0
       WHILE x<100
         WriteF('x is \d\n', x++)
       ENDWHILE
       WriteF('Finished\n')
```

1.101 beginner.guide/Unification

Unification
===========

In E, unification is a way of doing complicated, conditional assignments. It may also be referred to as pattern matching because that is what it does: it matches patterns and tries to fits values to the variables mentioned in those patterns. The result of a unification is true or false, depending on whether the pattern was successfully matched.

The basic form of a unification expression is:

```
       expression <=> pattern
```

The only things that can be used in a pattern are constants and variable names, and lists of patterns. (Strictly speaking, lisp-cells are also allowed, but this variant of unification is beyond the scope of this Guide.) The pattern is matched against the expression as follows:

* If pattern is a constant then the match succeeds only if expression evaluates to the same value. So, the simple unification expression x<=>1 is similar to an equality check x=1.

* If pattern is a variable name then the match is successful and the variable is assigned the value of expression. So, the simple unification expression 1<=>x is similar to an assignment x:=1.

* If pattern is a list then expression is assumed to be a list, and each element of pattern is taken to be a pattern to be (recursively) matched against the corresponding element (by index) of the expression list. The match succeeds only if the pattern list and the expression list are the same length and all the elements match. (It is a serious programming error if pattern is a list but expression does not represent a list. In this case, strange things may happen and the program may crash.)

So, the things in pattern that control whether a match succeeds are the constants and the lists.

If a match succeeds then all variables mentioned in the pattern will be assigned the appropriate values. However, if a match fails you should consider all variables involved in the pattern to have undefined values (so you may need to initialise them to safely their values again). This is because the actual way that unification is implemented may not follow the rules above in the obvious way, but will have the same effect in the successful case and will affect only the variables mentioned in the pattern if the match fails.

For example, the following program shows a couple of simple unification expressions in use:

```
PROC main()
  DEF x, lt
  x:=0
  WriteF('x is \d\n', x)
  lt:=[9,-1,7,4]

  /* The next line uses unification */
  IF lt <=> [9,-1,x,4]
    WriteF('First match succeeded\n')
    WriteF('1) x is now \d\n', x)
  ELSE
    WriteF('First match failed\n')
    /* To be safe, reset x */
    x:=0
  ENDIF

  /* The next line uses unification */
  IF lt <=> [1,x,6,4]
    WriteF('Second match succeeded\n')
    WriteF('2) x is now \d\n', x)
  ELSE
    WriteF('Second match failed\n')
```

```
    /* To be safe, reset x */
    x:=0
  ENDIF
ENDPROC
```

The first match will succeed in this example, and there will be a side-effect of assigning seven to x. The second match will not succeed because, for instance, the first element of lt is not one.

We can rewrite the above example without using the unification operator (to show why unification is so useful). This code follows the rules in one particular way, so is not guaranteed to have the same effect as the unification version if any of the matches fail.

```
PROC main()
  DEF x, lt, match
  x:=0
  WriteF('x is \d\n', x)
  lt:=[9,-1,7,4]

  /* The next lines mimic: lt <=> [9,-1,x,4] */
  match:=FALSE
  IF ListLen(lt)=4
    IF ListItem(lt, 0)=9
      IF ListItem(lt, 1)=-1
        x:=ListItem(lt,2)
        IF ListItem(lt, 3)=4
          match:=TRUE
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  IF match
    WriteF('First match succeeded\n')
    WriteF('1) x is now \d\n', x)
  ELSE
    WriteF('First match failed\n')
    /* To be safe, reset x */
    x:=0
  ENDIF

  /* The next lines mimic: lt <=> [1,x,6,4] */
  match:=FALSE
  IF ListLen(lt)=4
    IF ListItem(lt, 0)=1
      x:=ListItem(lt, 1)
      IF ListItem(lt, 2)=6
        IF ListItem(lt, 3)=4
          match:=TRUE
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  IF match
    WriteF('Second match succeeded\n')
    WriteF('2) x is now \d\n', x)
  ELSE
```

```
        WriteF('Second match failed\n')
        /* To be safe, reset x */
        x:=0
      ENDIF
    ENDPROC
```

Here's a slightly more complicated example, which shows how you might use patterns made up of nested lists. Remember that if the pattern is a list then the expression to be matched must be a list. If this is not the case (e.g., if the expression represents NIL) then your program could behave strangely or even crash your computer. A similar, but less disastrous, problem is if the converse happens: the pattern is not a list but the expression to be matched is a list. In this case the pointer (to the list) is matched against the pattern constant or assigned to the pattern variable.

```
    PROC main()
      DEF x=10, y=-3, p=NIL:PTR TO LONG, lt, i
      WriteF('x is \d, y is \d\n', x, y)
      lt:=[[23,x],y]

      /* This basically swaps x and y */
      IF lt <=> [[23,y],x]
        WriteF('First match succeeded\n')
        WriteF('1) Now x is \d, y is \d\n', x, y)
      ELSE
        WriteF('First match failed\n')
        /* To be safe, reset x and y */
        x:=10;  y:=-3
      ENDIF

      /* This will make p point to the sub-list of lt */
      IF lt <=> [p,-3]
        WriteF('Second match succeeded\n')
        WriteF('2) p is now $\h (a pointer to a list)\n', p)
        FOR i:=0 TO ListLen(p)-1
          WriteF('   Element \d of the list p is \d\n', i, p[i])
        ENDFOR
      ELSE
        WriteF('First match failed\n')
        /* To be safe, reset p */
        p:=NIL
      ENDIF
    ENDPROC
```

1.102 beginner.guide/Quoted Expressions

```
                Quoted Expressions
==================
```

Quoted expressions are a powerful feature of the E language, and they require quite a bit of advanced knowledge. Basically, you can quote any expression by starting it with the back-quote character ` (be careful not to get it mixed up with the quote character ' which is used for strings).

This quoting action does not evaluate the expression, instead the address of the code for the expression is returned. This address can be used just like any other address, so you can, for instance, store it in a variable and pass it to procedures. Of course, at some point you will use the address to execute the code and get the value of the expression.

The idea of quoted expressions was borrowed from the functional programming language Lisp. Also borrowed were some powerful functions which combine lists with quoted expressions to give very concise and readable statements.

1.103 beginner.guide/Evaluation

Evaluation

When you’ve quoted an expression you have the address of the code which calculates the value of the expression. To evaluate the expression you pass this address to the Eval function. So now we have a round-about way of calculating the value of an expression. (If you have a GB keyboard you can get the ‘ character by holding down the ALT key and pressing the ’ key, which is in the corner just below the ESC key. On a US and most European keyboards it’s on the same key but you don’t have to press the ALT key at the same time.)

```
PROC main()
  DEF adr, x, y
  x:=0; y:=0
  adr:=‘1+(fred(x,1)*8)-y
  x:=2; y:=7
  WriteF(’The value is \d\n’, Eval(adr))
  x:=1; y:=100
  WriteF(’The value is now \d\n’, Eval(adr))
ENDPROC

PROC fred(a,b) RETURN (a+b)*a+20
```

This is the output that should be generated:

```
The value is 202
The value is now 77
```

This example shows a quite complicated expression being quoted. The address of the expression is stored in the variable adr, and the expression is evaluated using Eval in the calls to WriteF. The values of the variables x and y when the expression is quoted are irrelevant--only their values each time Eval is used are significant. Therefore, when Eval is used in the second call to WriteF the values of x and y have changed so the resulting value is different.

Repeatedly evaluating the same expression is the most obvious use of quoted expressions. Another common use is when you want to do the same thing for a variety of different expressions. For example, if you wanted to discover the amount of time it takes to calculate the results of various expressions it would be best to use quoted expressions, something like this:

```
DEF x,y

PROC main()
  x:=999; y:=173
  time(`x+y,     'Addition')
  time(`x*y,     'Multiplication')
  time(`fred(x), 'Procedure call')
ENDPROC

PROC time(exp, message)
  WriteF(message)
  /* Find current time */
  Eval(exp)
  /* Find new time and calculate difference, t */
  WriteF(': time taken \d\n', t)
ENDPROC
```

This is just the outline of a program--it's not complete so don't bother running it. The complete version is given as a worked example later (see

Timing Expressions
).

1.104 beginner.guide/Quotable expressions

Quotable expressions

There is no restriction on the kinds of expression you can quote, except that you need to be careful about variable scoping. If you use local variables in a quoted expression you can only Eval it within the same procedure (so the variables are in scope). However, if you use only global variables you can Eval it in any procedure. Therefore, if you are going to pass a quoted expression to a procedure and do something with it, you should use only global variables.

A word of warning: Eval does not check to see if the address it's been given is really the address of an expression. You can therefore get in a real mess if you pass it, say, the address of a variable using {var }. You need to check all uses of things like Eval yourself, because the E compiler lets you write things like Eval(x+9), where you probably meant to write Eval(`x+9). That's because you might want the address you pass to

Eval to be the result of complicated expressions. So you may have meant to pass x+9 as the parameter!

1.105 beginner.guide/Lists and quoted expressions

Lists and quoted expressions

There are several E built-in functions which use lists and quoted expressions in powerful ways. These functions are similar to functional programming constructs and, basically, they allow for very readable code, which otherwise would require iterative algorithms (i.e., loops).

MapList(address,list,e-list,quoted-exp)

The address is the address of a variable (e.g., {x}), list is a list or E-list (the source), e-list is an E-list variable (the destination), and quoted-exp is the address of an expression which involves the addressed variable (e.g., ‘x+2). The effect of the function is to take, in turn, a value from list, store it at address, evaluate the quoted expression and store the result in the destination list. The resulting list is also returned (for convenience).

For example:

```
MapList({y}, [1,2,3,a,99,1+c], lt, ‘y*y)
```

results in lt taking the value:

```
[1,4,9,a*a,9801,(1+c)*(1+c)]
```

Functional programmers would say that MapList mapped the function (the quoted expression) across the (source) list.

ForAll(address,list,quoted-exp)

Works just like MapList except that the resulting list is not stored. Instead, ForAll returns TRUE if every element of the resulting list is TRUE (i.e., non-zero), and FALSE otherwise. In this way it decides whether the quoted expression is TRUE for all elements of the source list. For example, the following are TRUE:

```
ForAll({x}, [1,2,-13,8,0], ‘x<10)
ForAll({x}, [1,2,-13,8,0], ‘x<=8)
ForAll({x}, [1,2,-13,8,0], ‘x>-20)
```

and these are FALSE:

```
ForAll({x}, [1,2,-13,8,0], ‘x OR x)
ForAll({x}, [1,2,-13,8,0], ‘x=2)
ForAll({x}, [1,2,-13,8,0], ‘x<>2)
```

Exists(address,list,quoted-exp)

Works just like ForAll except it returns TRUE if the quoted expression is TRUE (i.e., non-zero) for at least one of the source list elements and FALSE otherwise. For example, the following are TRUE:

```
Exists({x}, [1,2,-13,8,0], ‘x<10)
Exists({x}, [1,2,-13,8,0], ‘x=2)
Exists({x}, [1,2,-13,8,0], ‘x>0)
```

and these are FALSE:

```
Exists({x}, [1,2,-13,8,0], ‘x<-20)
Exists({x}, [1,2,-13,8,0], ‘x=4)
Exists({x}, [1,2,-13,8,0], ‘x>8)
```

SelectList(address,list,e-list,quoted-exp)

Works just like MapList except the quoted-exp is used to decide which elements from list are copied to e-list. Only the elements for which quoted-exp evaluates to a non-zero (i.e., true) value are copied. The resulting list is also returned (for convenience).

For example:

```
SelectList({y}, [99,6,1,2,7,1,1,6,6], lt, ‘y>5)
```

results in lt taking the value:

```
[99,6,7,6,6]
```

1.106 beginner.guide/Assembly Statements

Assembly Statements

The E language incorporates an assembler so you can write Assembly mnemonics as E statements. You can even write complete Assembly programs and compile them using the E compiler. More powerfully, you can use E variables as part of the mnemonics, so you can really mix Assembly statements with normal E statements.

This is not really the place to discuss Assembly programming, so if you plan to use this feature of E you should get yourself a good book, preferably on Amiga Assembly. Remember that the Amiga uses the Motorola 68000 CPU, so you need to learn the Assembly language for that CPU. More powerful and newer Amigas use more advanced CPUs (such as the 68020) which have extra mnemonics. Programs written using just 68000 CPU mnemonics will work on all Amigas.

If you don’t know 68000 Assembly language you ought to skip this section and just bear in mind that E statements you don’t recognise are probably Assembly mnemonics.

Assembly and the E language

Static memory

Things to watch out for

1.107 beginner.guide/Assembly and the E language

Assembly and the E language

You can reference E variables simply by using them in an operand. Follow the comments in the next example (the comments are on the same lines as the Assembly mnemonics, the other lines are normal E statements):

```
PROC main()
  DEF x
  x:=1
  MOVE.L x,  D0 /* Copy the value in x to register D0      */
  ADD.L  D0, D0 /* Double the value in D0                  */
  MOVE.L D0, x  /* Copy the value in D0 back to variable x */
  WriteF('x is now \d\n', x)
ENDPROC
```

Constants can also be referenced but you need to precede the constant with a #.

```
CONST APPLE=2

PROC main()
  DEF x
  MOVE.L #APPLE, D0 /* Copy the constant APPLE to register D0 */
  ADD.L  D0, D0     /* Double the value in D0                 */
  MOVE.L D0, x      /* Copy the value in D0 to variable x     */
  WriteF('x is now \d\n', x)
ENDPROC
```

Labels and procedures can similarly be referenced, but these are PC-relative so you can only address them in this way. The following example illustrates this, but doesn't do anything useful:

```
PROC main()
  DEF x
  LEA main(PC), A0 /* Copy the address of main to register A0 */
  MOVE.L A0, x     /* Copy the value in A0 to variable x      */
  WriteF('x is now \d\n', x)
ENDPROC
```

You can call Amiga system functions in the same way as you would normally in Assembly. You need to load the A6 register with the appropriate library base, load the other registers with appropriate data and then JSR to the system routine. This next example uses the E built-in variable intuitionbase and the Intuition library routine DisplayBeep. When you run it the screen flashes (or, with Workbench 2.1 and above, you might get a beep or a sampled sound, depending on your Workbench setup).

```
PROC main()
  MOVE.L #NIL, A0
  MOVE.L intuitionbase, A6
  JSR DisplayBeep(A6)
ENDPROC
```

1.108 beginner.guide/Static memory

Static memory

Assembly programs reserve static memory for things like strings using DC mnemonics. However, these aren’t real mnemonics. They are, in fact, compiler directives and they can vary from compiler to compiler. The E versions are LONG, INT and CHAR (the type names), which take a list of values, reserve the appropriate amount of static memory and fill in this memory with the supplied values. The CHAR form also allows a list of characters to be supplied more easily as a string. In this case, the string will automatically be aligned to an even memory location, although you are responsible for null-terminating it. You can also include a whole file as static data using INCBIN (and the file named using this statement must exist when the program is compiled). To get at the data you mark it with a label.

This next example is a bit contrived, but illustrates some static data:

```
PROC main()
  DEF x:PTR TO CHAR
  LEA datatable(PC), A0
  MOVE.L A0, x
  WriteF(x)
ENDPROC

datatable:
  CHAR ’Hello world\n’, 0
moredata:
  LONG 1,5,-999,0;    INT -1,222
  INCBIN ’file.data’; CHAR 0,7,-8
```

The Assembly stuff to get the label address is not really necessary, so the example could have been just:

```
PROC main()
  WriteF({datatable})
ENDPROC

datatable:
  CHAR ’Hello world\n’, 0
```

1.109 beginner.guide/Things to watch out for

Things to watch out for

There are a few things to be aware of when using Assembly with E:

- All mnemonics and registers must be in uppercase, whilst, of course, E variables etc., follow the normal E rules.

- Most standard Assemblers use ; to mark the rest of the line as a comment. In E you can use -> for the same effect, or you can use the /* and */ delimiters.

- Registers A4 and A5 are used internally by E, so should not be messed with if you are mixing E and Assembly code. Other registers might also be used, especially if you've used the REG keyword. Refer to the 'Reference Manual' for more details.

- E function calls like WriteF can affect registers. Refer to the 'Reference Manual' for more details.

1.110 beginner.guide/E Built-In Constants Variables and Functions

E Built-In Constants, Variables and Functions
****

This chapter describes the constants, variables and functions which are built-in to the E language. You can add more by using modules, but that's a more advanced topic (see
Modules
).

Built-In Constants

Built-In Variables

Built-In Functions

1.111 beginner.guide/Built-In Constants

Built-In Constants
==================

We've already met several built-in constants. Here's the complete list:

TRUE, FALSE
The boolean constants. As numbers, TRUE is -1 and FALSE is zero.

NIL
     The bad pointer value. Several functions produce this value for a pointer if an error occurred. As a number, NIL is zero.

ALL
     Used with string and list functions to indicate that all the string or list is to be used. As a number, ALL is -1.

GADGETSIZE
     The minimum number of bytes required to hold all the data for one gadget. See
           Intuition support functions
           .

OLDFILE, NEWFILE
     Used with Open to open an old or new file. See the ‘AmigaDOS Manual’ for more details.

STRLEN
     The length of the last string constant used. Remember that a string constant is something between ’ characters, so, for example, the following program prints the string s and then its length:

```
PROC main()
  DEF s:PTR TO CHAR, len
  s:='12345678’
  len:=STRLEN
  WriteF(s)
  WriteF(’\nis \d characters long\n’, len)
ENDPROC
```

1.112 beginner.guide/Built-In Variables

```
                Built-In Variables
==================
```

The following variables are built-in to E and are called system variables. They are global so can be accessed from any procedure.

arg
     This is a string which contains the command line arguments passed your program when it was run (from the Shell or CLI). For instance, if your program were called fred and you ran it like this:

```
fred file.txt "a big file" another
```

then arg would the string:

```
file.txt "a big file" another
```

If you have AmigaDOS 2.0 (or greater) you can use the system routine ReadArgs to parse the command line in a much more versatile way. There is a worked example on argument parsing in Part Three (see

Argument Parsing
).

wbmessage
This contains NIL if your program was started from the Shell/CLI, otherwise it’s a pointer to the Workbench message which contains information about the icons selected when you started the program from Workbench. So, if you started the program from Workbench wbmessage will not be NIL and it will contain the Workbench arguments, but if you started the program from the Shell/CLI wbmessage will be NIL and the argments will be in arg (or via ReadArgs). There is a worked example on argument parsing in Part Three (see
Argument Parsing
).

stdin, stdout, conout
The stdin and stdout variables contain the standard input and output filehandles. If your program was started from the Shell/CLI they will be filehandles on the Shell/CLI window (and conout will be NIL). However, if your program was started from Workbench these will both be NIL, and in this case the first call to WriteF will open an output CON: window and store the file handle for the window in stdout and conout. The file handle stored in conout when the program terminates will be closed using Close, so you can set up your own CON: window or file for use by the output functions and have it automatically closed. See
Input and output functions
.

stdrast
The raster port used by E built-in graphics functions such as Box and Plot. This can be changed so that these functions draw on different screens etc. See
Graphics functions
.

dosbase, execbase, gfxbase, intuitionbase
These are pointers to the appropriate library base, and are initialised by the E startup code, i.e., the Dos, Exec, Graphics and Intuition libraries are all opened by E so you don’t need to do it yourself. These libraries are also automatically closed by E, so you shouldn’t close them yourself. However, you must explicitly open and close all other Amiga system libraries that you want to use. The other library base variables are defined in the accompanying module (see
Modules
). Consult the ‘Reference Manual’ for details about the library base variable mathbase.

1.113 beginner.guide/Built-In Functions

Built-In Functions

There are many built-in functions in E. We've already seen a lot of string and list functions, and we've used WriteF for printing. The remaining functions are, generally, simplifications of complex Amiga system functions, or E versions of support functions found in languages like C and Pascal.

To understand the graphics and Intuition support functions completely you really need to get something like the 'Rom Kernel Reference Manual (Libraries)'. However, if you don't want to do anything too complicated you can just about get by.

Input and output functions

Intuition support functions

Graphics functions

Maths and logic functions

System support functions

1.114 beginner.guide/Input and output functions

Input and output functions

WriteF(string,param1,param2,...)

Writes a string to the standard output and returns the number of characters written. If place-holders are used in the string then the appropriate number of parameters must be supplied after the string in the order they are to be printed as part of the string. So far we've only met the \d place-holder for decimal numbers. The complete list is:

Place-Holder	Parameter Type	Prints
\c	Number	Character
\d	Number	Decimal number
\h	Number	Hexadecimal number
\s	String	String

So to print a string you use the \s place-holder in the string and supply the string (e.g., a PTR TO CHAR) as a parameter. Try the following program (remember \a prints an apostrophe character):

```
PROC main()
  DEF s[30]:STRING
  StrCopy(s, 'Hello world', ALL)
```

```
  WriteF('The third element of s is "\c"\n', s[2])
  WriteF('or \d (decimal)\n',                s[2])
  WriteF('or \h (hexadecimal)\n',            s[2])
  WriteF('and s itself is \a\s\a\n',         s)
ENDPROC
```

This is the output it generates:

```
The third element of s is "l"
or 108 (decimal)
or 6C (hexadecimal)
and s itself is 'Hello world'
```

You can control how the parameter is formatted in the \d, \h and \s fields using another collection of special character sequences before the place-holder and size specifiers after it. If no size is specified the field will be as big as the data requires. A fixed field size can be specified using [number] after the place-holder. For strings you can also use the size specifier (min,max) which specifies the minimum and maximum sizes of the field. By default the data is right justified in the field and the left part of the field is filled, if necessary, with spaces. The following sequences before the place-holder can change this:

Sequence	Meaning
\l	Left justify in field
\r	Right justify in field
\z	Set fill character to "0"

See how these formatting controls affect this example:

```
PROC main()
  DEF s[30]:STRING
  StrCopy(s, 'Hello world', ALL)
  WriteF('The third element of s is "\c"\n', s[2])
  WriteF('or \d[4] (decimal)\n',             s[2])
  WriteF('or \z\h[4] (hexadecimal)\n',       s[2])
  WriteF('\a\s[5]\a are the first five elements of s \n', s)
  WriteF('and s in a very big field  \a\s[20]\a\n',   s)
  WriteF('and s left justified in it \a\l\s[20]\a\n', s)
ENDPROC
```

Here's the output it should generate:

```
The third element of s is "l"
or  108 (decimal)
or 006C (hexadecimal)
'Hello' are the first five elements of s
and s in a very big field  '         Hello world'
and s left justified in it 'Hello world         '
```

WriteF uses the standard output, and this file handle is stored in the stdout variable. If your program is started from Workbench this variable will contain NIL. In this case, the first call to WriteF will open a special output window and put the file handle in the variables stdout and conout, as outlined above.

PrintF(string,param1,param2,...)

PrintF works just like WriteF except it uses the more efficient, buffered output routines only available if your Amiga is using Kickstart version 37 or greater (i.e., AmigaDOS 2.04 and above).

StringF(e-string,string,arg1,arg2,...)

The same as WriteF except that the result is written to e-string instead of being printed. For example, the following code fragment sets s to 00123 is a (since the E-string is not long enough for the whole string):

```
DEF s[10]:STRING
StringF(s, ’\z\d[5] is a number’, 123)
```

Out(filehandle,char)

Outputs a single character, char, to the file or console window denoted by filehandle, and returns -1 to indicate success (so any other return value means an error occurred). For instance, filehandle could be stdout, in which case the character is written to the standard output. (You need to make sure stdout is not NIL, and you can do this by using a WriteF(’’) call.)

Inp(filehandle)

Reads and returns a single character from filehandle. If -1 is returned then the end of the file (EOF) was reached, or there was an error.

ReadStr(filehandle,e-string)

Reads a whole string from filehandle and returns -1 if EOF was reached or an error occurred. Characters are read up to a linefeed or the size of the string, which ever is sooner. Therefore, the resulting string may be only a partial line. If -1 is returned then EOF was reached or an error occurred, and in either case the string so far is still valid. So, you still need to check the string even if -1 is returned. (This will most commonly happen with files that do not end with a linefeed.) The string will be empty (i.e., of zero length) if nothing more had been read from the file when the error or EOF happened.

This next little program reads continually from its input until an error occurs or the user types quit. It echoes the lines that it reads in uppercase. If you type a line longer than ten characters you’ll see it reads it in more than one go. Because of the way normal console windows work, you need to type a return before a line gets read by the program (but this allows you to edit the line before the program sees it). If the program is started from Workbench then stdin would be NIL, so WriteF(’’) is used to force stdout to be valid, and in this case it will be a new console window which can be used to accept input! (To make the compiled program into a Workbench program you simply need to create a tool icon for it. A quick way of doing this is to copy an existing tool’s icon.)

```
PROC main()
  DEF s[10]:STRING, fh
  WriteF(’’)
  fh:=IF stdin THEN stdin ELSE stdout
```

```
           WHILE ReadStr(fh, s)<>-1
             UpperStr(s)
           EXIT StrCmp(s, 'QUIT', ALL)
             WriteF('Read: \a\s\a\n', s)
           ENDWHILE
           WriteF('Finished\n')
         ENDPROC
```

There are some worked examples in Part Three (see

String Handling and I-O
) which also show how to use ReadStr.

FileLength(string)
Returns the length of the file named in string, or -1 if the file doesn't exist or an error occurred. Notice that you don't need to Open the file or have a filehandle, you just supply the filename. There is a worked example in Part Three (see
String Handling and I-O
)
which shows how to use this function.

SetStdIn(filehandle)
Returns the value of stdin before setting it to filehandle. Therefore, the following code fragments are equivalent:

```
oldstdin:=SetStdIn(newstdin)

oldstdin:=stdin
stdin:=newstdin
```

SetStdOut(filehandle)
Returns the value of stdout before setting it to filehandle, and is otherwise just like SetStdIn.

1.115 beginner.guide/Intuition support functions

Intuition support functions

The functions in this section are simplified versions of Amiga system functions (in the Intuition library, as the title suggests). To make best use of them you are probably going to need something like the 'Rom Kernel Reference Manual (Libraries)', especially if you want to understand the Amiga specific things like IDCMP and raster ports.

The descriptions given here vary slightly in style from the previous descriptions. All function parameters can be expressions which represent numbers or addresses, as appropriate. Because many of the functions take several parameters they have been named (fairly descriptively) so they can be more easily referenced.

OpenW(x,y,wid,hgt,idcmp,wflgs,title,scrn,sflgs,gads,tags=NIL)

Opens and returns a pointer to a window with the supplied properties. If for some reason the window could not be opened NIL is returned.

x, y

The position on the screen where the window will appear.

wid, hgt

The width and height of the window.

idcmp, wflgs

The IDCMP and window specific flags.

title

The window title (a string) which appears on the title bar of the window.

scrn, sflgs

The screen on which the window should open. If sflgs is 1 the window will be opened on Workbench, and scrn is ignored (so it can be NIL). If sflgs is $F (i.e., 15) the window will open on the custom screen pointed to by scrn (which must then be valid). See OpenS to see how to open a custom screen and get a screen pointer.

gads

A pointer to a gadget list, or NIL if you don't want any gadgets. These are not the standard window gadgets, since they are specified using the window flags. A gadget list can be created using the Gadget function.

tags

A tag-list of other options available under Kickstart version 37 or greater. This can normally be omitted since it defaults to NIL. See the 'Rom Kernel Reference Manual (Libraries)' for details about the available tags and their meanings.

There's not enough space to describe all the fine details about windows and IDCMP (see the 'Rom Kernel Reference Manual (Libraries)' for complete details), but a brief description in terms of flags might be useful. Here's a small table of common IDCMP flags:

IDCMP Flag	Value
IDCMP_NEWSIZE	$2
IDCMP_REFRESHWINDOW	$4
IDCMP_MOUSEBUTTONS	$8
IDCMP_MOUSEMOVE	$10
IDCMP_GADGETDOWN	$20
IDCMP_GADGETUP	$40
IDCMP_MENUPICK	$100
IDCMP_CLOSEWINDOW	$200
IDCMP_RAWKEY	$400
IDCMP_DISKINSERTED	$8000
IDCMP_DISKREMOVED	$10000

Here's a table of useful window flags:

Window Flag	Value
WFLG_SIZEGADGET	$1
WFLG_DRAGBAR	$2
WFLG_DEPTHGADGET	$4
WFLG_CLOSEGADGET	$8
WFLG_SIZEBRIGHT	$10
WFLG_SIZEBBOTTOM	$20
WFLG_SMART_REFRESH	0
WFLG_SIMPLE_REFRESH	$40
WFLG_SUPER_BITMAP	$80
WFLG_BACKDROP	$100
WFLG_REPORTMOUSE	$200
WFLG_GIMMEZEROZERO	$400
WFLG_BORDERLESS	$800
WFLG_ACTIVATE	$1000

All these flags are defined in the module intuition/intuition, so if you use that module you can use the constants rather than having to write the less descriptive value (see Modules). Of course, you can always define your own constants for the values that you use.

You use the flags by OR-ing the ones you want together, in similar way to using sets (see Sets). However, you should supply only IDCMP flags as part of the idcmp parameter, and you should supply only window flags as part of the wflgs parameter. So, to get IDCMP messages when a disk is inserted and when the close gadget is clicked you specify both of the flags IDCMP_DISKINSERTED and IDCMP_CLOSEWINDOW for the idcmp parameter, either by OR-ing the constants or (less readably) by using the calculated value $8200.

Some of the window flags require some of IDCMP flags to be used as well, if an effect is to be complete. For example, if you want your window to have a close gadget (a standard window gadget) you need to use WFLG_CLOSEGADGET as one of the window flags. If you want that gadget to be useful then you need to get an IDCMP message when the gadget is clicked. You therefore need to use IDCMP_CLOSEWINDOW as one of the IDCMP flags. So the full effect requires both a window and an IDCMP flag (a gadget is pretty useless if you can't tell when it's been clicked). The worked example in Part Three illustrates how to use these flags in this way (see Gadgets).

If you only want to output text to a window (and maybe do some input from a window), it may be better to use a console window. These provide a text based input and output window, and are opened using the Dos library function Open with the appropriate CON: file name. See the 'AmigaDOS Manual' for more details about console windows.

CloseW(winptr)

Closes the window which is pointed to by winptr. It's safe to give NIL for winptr, but in this case, of course, no window will be closed! The window pointer is usually a pointer returned by a matching call to OpenW. You must remember to close any windows you may have opened before terminating your program.

OpenS(wid,hgt,depth,scrnres,title,tags=NIL)

Opens and returns a pointer to a custom screen with the supplied properties. If for some reason the screen could not be opened NIL is returned.

wid, hgt

The width and height of the screen.

depth

The depth of the screen, i.e., the number of bit-planes. This can be a number in the range 1-8 for AGA machines, or 1-6 for pre-AGA machines. A screen with depth 3 will be able to show 2 to the power 3 (i.e., 8) different colours, since it will have 2 to the power 3 different pens (or colour registers) available. You can set the colours of pens using the SetColour function.

scrnres

The screen resolution flags.

title

The screen title (a string) which appears on the title bar of the screen.

tags

A tag-list of other options available under Kickstart version 37 or greater. See the ‘Rom Kernel Reference Manual (Libraries)’ for more details.

The screen resolution flags control the screen mode. The following (common) values are taken from the module graphics/view (see Modules). You can, if you want, define your own constants for the values that you use. Either way it’s best to use descriptive constants rather than directly using the values.

Mode Flag	Value
V_LACE	$4
V_SUPERHIRES	$20
V_PFBA	$40
V_EXTRA_HALFBRITE	$80
V_DUALPF	$400
V_HAM	$800
V_HIRES	$8000

So, to get a hires, interlaced screen you specify both of the flags V_HIRES and V_LACE, either by OR-ing the constants or (less readably) by using calculated value $8004. There is a worked example using this function in Part Three (see Screens).

CloseS(scrnptr)

Closes the screen which is pointed to by scrnptr. It's safe to give NIL for scrnptr, but in this case, of course, no screen will be closed! The screen pointer is usually a pointer returned by a matching call to OpenS. You must remember to close any screens you may have opened before terminating your program. Also, you must close all windows that you opened on your screen before you can close the screen.

Gadget(buf,glist,id,flags,x,y,width,text)

Creates a new gadget with the supplied properties and returns a pointer to the next position in the (memory) buffer which can be used for a gadget.

buf

This is the memory buffer, i.e., a chunk of allocated memory. The best way of allocating this memory is to declare an array of size n*GADGETSIZE, where n is the number of gadgets which are going to be created. The first call to Gadget will use the array as the buffer, and subsequent calls use the result of the previous call as the buffer (since this function returns the next free position in the buffer).

glist

This is a pointer to the gadget list that is being created, i.e., the array used as the buffer. When you create the first gadget in the list using an array a, this parameter should be NIL. For all other gadgets in the list this parameter should be the array a.

id

A number which identifies the gadget. It is best to give a unique number for each gadget, that way you can easily identify them. This number is the only way you can identify which gadget has been clicked.

flags

The type of gadget to be created. Zero represents a normal gadget, one a boolean gadget (a toggle) and three a boolean that starts selected.

x, y

The position of the gadget, relative to the top, left-hand corner of the window.

width

The width of the gadget (in pixels, not characters).

text

The text (a string) which will centred in the gadget, so the width must be big enough to hold this text.

Once a gadget list has been created by possibly several calls to this function the list can be passed as the gads parameter to OpenW. There is a worked example using this function in Part Three (see

Gadgets

).

Mouse()

Returns the state of the mouse buttons (including the middle mouse button if you have a three-button mouse). This is a set of flags, and the individual flag values are:

Button Pressed	Value
Left	%001
Right	%010
Middle	%100

So, if this function returns %001 you know the left button is being pressed, and if it returns %110 you know the middle and right buttons are both being pressed.

This mouse function is not strictly the proper way to do things. It is suggested you use this function only for small tests or demo-like programs. The proper way of getting mouse details is to use the appropriate IDCMP flags for your window, wait for events and decode the information.

MouseX(winptr)

Returns the x coordinate of the mouse pointer, relative to the window pointed to by winptr.

This mouse function is not strictly the proper way to do things. It is suggested you use this function only for small tests or demo-like programs. The proper way of getting mouse details is to use the appropriate IDCMP flags for your window, wait for events and decode the information.

MouseY(winptr)

Returns the y coordinate of the mouse pointer, relative to the window pointed to by winptr.

This mouse function is not strictly the proper way to do things. It is suggested you use this function only for small tests or demo-like programs. The proper way of getting mouse details is to use the appropriate IDCMP flags for your window, wait for events and decode the information.

LeftMouse(winptr)

Returns TRUE if left mouse button has been clicked in the window pointed to by winptr, and FALSE otherwise. In order for this to work sensibly the window must have the IDCMP flag IDCMP_MOUSEBUTTONS set (see above).

This function does things in a proper, Intuition-friendly manner and so is a good alternative to the Mouse function.

WaitIMessage(winptr)

This function waits for a message from Intuition for the window pointed to by winptr and returns the class of the message (which is an IDCMP flag). If you did not specify any IDCMP flags when the window was opened, or the specified messages could never happen (e.g., you asked only for gadget messages and you have no gadgets), then this function may wait forever. When you've got a message you can use the MsgXXX functions to get some more information about the message. See the 'Rom Kernel Reference Manual (Libraries)' for more details on Intuition and IDCMP. There is a worked example using this function in Part Three (see

IDCMP Messages

).

This function is basically equivalent to the following function, except that the MsgXXX functions can also access the message data held in the variables code, qual and iaddr.

```
PROC waitimessage(win:PTR TO window)
  DEF port,msg:PTR TO intuimessage,class,code,qual,iaddr
  port:=win.userport
  IF (msg:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (msg:=GetMsg(port))<>NIL
  ENDIF
  class:=msg.class
  code:=msg.code
  qual:=msg.qualifier
  iaddr:=msg.iaddress
  ReplyMsg(msg)
ENDPROC class
```

MsgCode()

Returns the code part of the message returned by WaitIMessage.

MsgIaddr()

Returns the iaddr part of the message returned by WaitIMessage. There is a worked example using this function in Part Three (see

IDCMP Messages

).

MsgQualifier()

Returns the qual part of the message returned by WaitIMessage.

WaitLeftMouse(winptr)

This function waits for the left mouse button to be clicked in the window pointed to by winptr. It is advisable to have the IDCMP flag IDCMP_MOUSEBUTTONS set for the window (see above).

This function does things in a proper, Intuition-friendly manner and so is a good alternative to the Mouse function.

1.116 beginner.guide/Graphics functions

Graphics functions

The functions in this section use the standard raster port, the address of which is held in the variable stdrast. Most of the time you don't need to worry about this because the E functions which open windows and screens set up this variable (see Intuition support functions). So, by default, these functions affect the last window or screen opened. When you close a window or screen, stdrast becomes NIL and calls to these functions have no effect. There is a worked example using these functions in Part Three (see Graphics).

The descriptions in this section follow the same style as the previous section.

Plot(x,y,pen=1)
: Plots a single point (x,y) in the specified pen colour. The position is relative to the top, left-hand corner of the window or screen that is the current raster port (normally the last screen or window to be opened). The range of pen values available depend on the screen setup, but are at best 0-255 on AGA machines and 0-31 on pre-AGA machines. As a guide, the background colour is usually pen zero, and the main foreground colour is pen one (and this is the default pen). You can set the colours of pens using the SetColour function.

Line(x1,y1,x2,y2,pen=1)
: Draws the line (x1,y1) to (x2,y2) in the specified pen colour.

Box(x1,y1,x2,y2,pen=1)
: Draws the (filled) box with vertices (x1,y1), (x2,y1), (x1,y2) and (x2,y2) in the specified pen colour.

Colour(fore-pen,back-pen=0)
: Sets the foreground and background pen colours. As mentioned above, the background colour is normally pen zero and the main foreground is pen one. You can change these defaults with this function, and if you stick to having the background pen as pen zero then calling this function with one argument changes just the foreground pen.

TextF(x,y,format-string,arg1,arg2,...)
: This works just like WriteF except the resulting string is written starting at point (x,y). Also, don't use any line-feed, carriage return, tab or escape characters in the string--they don't behave like they do in WriteF.

SetColour(scrnptr,pen,r,g,b)
: Sets the colour of colour register pen for the screen pointed to by scrnptr to be the appropriate RGB value (i.e., red value r, green value g and blue value b). The pen can be anything up to 255, depending on the screen depth. Regardless of the chipset being used, r, g and b are taken from the range zero to 255, so 24-bit colours are always specified. In operation, though, the values are scaled to 12-bit colour for non-AGA machines.

```
SetStdRast(newrast)
     Returns the value of stdrast before setting it to the new value.  The
     following code fragments are equivalent:

              oldstdrast:=SetStdRast(newstdrast)

              oldstdrast:=stdrast
              stdrast:=newstdrast

SetTopaz(size=8)
     Sets the text font for the current raster port to Topaz at the
     specified size, which defaults to the standard size eight.
```

1.117 beginner.guide/Maths and logic functions

Maths and logic functions

We've already seen the standard arithmetic operators. The addition, +, and subtraction, -, operators use full 32-bit integers, but, for efficiency, multiplication, *, and division, /, use restricted values. You can only use * to multiply 16-bit integers, and the result will be a 32-bit integer. Similarly, you can only use / to divide a 32-bit integer by a 16-bit integer, and the result will be a 16-bit integer. The restrictions do not affect most calculations, but if you really need to use all 32-bit integers (and you can cope with overflows etc.) you can use the Mul and Div functions. Mul(a,b) corresponds to a*b, and Div(a,b) corresponds to a/b.

We've also met the logic operators AND and OR, which we know are really bit-wise operators. You can also use the functions And and Or to do exactly the same as AND and OR (respectively). So, for instance, And(a,b) is the same as a AND b. The reason for these functions is because there are Not and Eor (bit-wise) functions, too (and there aren't operators for these). Not(a) swaps one and zero bits, so, for instance, Not(TRUE) is FALSE and Not(FALSE) is TRUE. Eor(a,b) is the exclusive version of Or and does almost the same, except that Eor(1,1) is 0 whereas Or(1,1) is 1 (and this extends to all the bits). So, basically, Eor tells you which bits are different, or, logically, if the truth values are different. Therefore, Eor(TRUE,TRUE) is FALSE and Eor(TRUE,FALSE) is TRUE.

There's a collection of other functions related to maths, logic or numbers in general:

```
Abs(expression)
     Returns the absolute value of expression.  The absolute value of a
     number is that number made positive if necessary.  So, Abs(9) is 9,
     and Abs(-9) is also 9.

Sign(expression)
     Returns the sign of expression, which is the value one if it is
     (strictly) positive, -1 if it is (strictly) negative and zero if it
```

is zero.

Even(expression)

Returns TRUE if expression represents an even number, and FALSE otherwise. Obviously, a number is either odd or even!

Odd(expression)

Returns TRUE if expression represents an odd number, and FALSE otherwise.

Max(exp1, exp2)

Returns the maximum of exp1 and exp2.

Min(exp1, exp2)

Returns the minimum of exp1 and exp2.

Bounds(exp, minexp, maxexp)

Returns the value of exp bounded to the limits minexp (minimum bound) and maxexp (maximum bound). That is, if exp lies between the bounds then exp is returned, but if it is less than minexp then minexp is returned or if it is greater than maxexp then maxexp is returned. This is useful for, say, constraining a calculated value to be a valid (integer) percentage (i.e., a value between zero and one hundred).

The following code fragments are equivalent:

```
y:=Bounds(x, min, max)

y:=IF x<min THEN min ELSE IF x>max THEN max ELSE x
```

Mod(exp1,exp2)

Returns the 16-bit remainder (or modulus) of the division of the 32-bit exp1 by the 16-bit exp2 as the regular return value (see

Multiple Return Values

), and the 16-bit result of the division as the first optional return value. For example, the first assignment in the following code sets a to 5 (since 26=(7*3)+5), b to 3, c to -5 and d to -3. It is important to notice that if exp1 is negative then the modulus will also be negative. This is because of the way integer division works: it simply discards fractional parts rather rounding.

```
a,b:=Mod(26,7)
c,d:=Mod(-26,7)
```

Rnd(expression)

Returns a random number in the range 0 to (n-1), where expression represents the value n. These numbers are pseudo-random, so although you appear to get a random value from each call, the sequence of numbers you get will probably be the same each time you run your program. Before you use Rnd for the first time in your program you should call it with a negative number. This decides the starting point for the pseudo-random numbers.

RndQ(expression)

Returns a random 32-bit value, based on the seed expression. This function is quicker than Rnd, but returns values in the 32-bit range, not a specified range. The seed value is used to select different sequences of pseudo-random numbers, and the first call to RndQ should use a large value for the seed.

Shl(exp1,exp2)
Returns the value represented by exp1 shifted exp2 bits to the left. For example, Shl(%0001110,2) is %0111000 and Shl(%0001011,3) is %1011000. Shifting a number one bit to the left is generally the same as multiplying it by two (although this isn't true when you shift large positive or large negative values). (The new bits shifted in at the right are always zeroes.)

Shr(exp1,exp2)
Returns the value represented by exp1 shifted exp2 bits to the right. For example, Shr(%0001110,2) is %0000011 and Shr(%1011010,3) is %0001011. Shifting a number one bit to the right is generally the same as dividing it by two. (The new bits shifted in at the left are always zeroes.)

Long(addr), Int(addr), Char(addr)
Returns the LONG, INT or CHAR value at the address addr. These functions should be used only when setting up a pointer and dereferencing it in the normal way would make your program cluttered and less readable. Use of functions like these is often called peeking memory (especially in dialects of the BASIC language).

PutLong(addr,exp), PutInt(addr,exp), PutChar(addr,exp)
Writes the LONG, INT or CHAR value represented by exp to the address addr. Again, these functions should be used only when really necessary. Use of functions like these is often called poking memory.

1.118 beginner.guide/System support functions

System support functions

New(bytes)
Returns a pointer to a newly allocated chunk of memory, which is bytes number of bytes. If the memory could not be allocated NIL is returned. The memory is initialised to zero in each byte, and taken from any available store (Fast or Chip memory, in that order of preference). When you've finished with this memory you can use Dispose to free it for use elsewhere in your program. You don't have to Dispose with memory you allocated with New because your program will automatically free it when it terminates. This is not true for memory allocated using the normal Amiga system routines.

NewR(bytes)
The same as New except that if the memory could not be allocated then the exception "MEM" is raised (and so, in this case, the function

does not return). See Exception Handling.

NewM(bytes,type)

The same as NewR except that the type of memory (Fast or Chip) to be allocated can be specified using flags. The flags are defined in the module exec/memory (see Amiga System Modules). See the 'Rom Kernel Reference Manual (Libraries)' for details about the system function AllocMem which uses these flags in the same way. As useful example, here's a small program which allocates some cleared (i.e., zeroed) Chip memory.

```
MODULE 'exec/memory'

PROC main()
  DEF m
  m:=NewM(20, MEMF_CHIP OR MEMF_CLEAR)
  WriteF('Allocation succeeded, m = $\h\n', m)
EXCEPT
  IF exception="NEW" THEN WriteF('Failed\n')
ENDPROC
```

Dispose(address)

Used to free memory allocated with New, NewR or NewM. You should rarely need to use this function because the memory is automatically freed when the program terminates.

DisposeLink(complex)

Used to free the memory allocated String (see String functions) and List (see List functions). Again, you should rarely need to use this function because the memory is automatically freed when the program terminates.

FastNew(bytes)

The same as NewR except it uses a very fast, recycling method of allocating memory. The memory allocated using FastNew is, as ever, deallocated automatically at the end of a program, and can be deallocated before then using FastDispose. Note that only FastDispose can be used and that it differs slightly from the Dispose and DisposeLink functions (you have to specify the number of bytes originally allocated when deallocating).

FastDispose(address,bytes)

Used to free the memory allocated using FastNew. The bytes parameter must be the same as the bytes used when allocating with FastNew, but the benefit is much faster allocation and deallocation and generally more efficient use of memory.

CleanUp(expression=0)

Terminates the program at this point, and does the normal things an E

```
     program does when it finishes.  The value denoted by expression is
     returned as the error code for the program.  It is the replacement
     for the AmigaDOS Exit routine which should never be used in an E
     program.  This is the only safe way of terminating a program, other
     than reaching the (logical) end of the main procedure (which is by
     far the most common way!).

CtrlC()
     Returns TRUE if control-C has been pressed since the last call, and
     FALSE otherwise.  This is only sensible for programs started from the
     Shell/CLI.

FreeStack()
     Returns the current amount of free stack space for the program.  Only
     complicated programs need worry about things like stack.  Recursion
     is the main thing that eats a lot of stack space.

KickVersion(expression)
     Returns TRUE if your Kickstart revision is at least that given by
     expression, and FALSE otherwise.  For instance, KickVersion(37)
     checks whether you're running with Kickstart version 37 or greater
     (i.e., AmigaDOS 2.04 and above).
```

1.119 beginner.guide/Modules

```
                Modules
*******

   A module is the E equivalent of a C header file and an Assembly
include file.  It can contain various object and constant definitions, and
also library function offsets and library base variables.  This information is
necessary for the correct use of a library.



                 Using Modules

                 Amiga System Modules

                 Non-Standard Modules

                 Example Module Use

                 Code Modules
```

1.120 beginner.guide/Using Modules

```
Using Modules
=============
```

To use the definitions in a particular module you use the MODULE statement at the beginning of your program (before the first procedure definition). You follow the MODULE keyword by a comma-separated list of strings, each of which is the filename (or path if necessary) of a module without the .m extension (every module file ends with .m). The filenames (and paths) are all relative to the logical volume Emodules:, which is set-up using an assign as described in the ‘Reference Manual’, unless the first character of the string is *. In this case the files are relative to the directory of the current source file. For instance, the statement:

```
     MODULE 'fred', 'dir/barney', '*mymod'
```

will try to load the files Emodules:fred.m, Emodules:dir/barney.m and mymod.m. If it can’t find these files or they aren’t proper modules the E compiler will complain.

All the definitions in the modules included in this way are available to every procedure in the program. To see what a module contains you can use the showmodule program that comes with the Amiga E distribution.

1.121 beginner.guide/Amiga System Modules

Amiga System Modules

Amiga E comes with the standard Amiga system include files as E modules. The AmigaDOS 2.04 modules are supplied with E version 2.1, and the AmigaDOS 3.0 modules are supplied with E version 3.0. However, modules are much more useful in E version 3.0 (see Code Modules). If you want to use any of the standard Amiga libraries properly you will need to investigate the modules for that library. The top-level .m files in Emodules: contain the library function offsets, and those in directories in Emodules: contain constant and object definitions for the appropriate library. For instance, the module asl (i.e., the file Emodules:asl.m) contains the ASL library function offsets and libraries/asl contains the ASL library constants and objects.

If you are going to use, say, the ASL library then you need to open the library using the OpenLibrary function (an Amiga system function) before you can use any of the library functions. You also need to define the library function offsets by using the MODULE statement. However, the DOS, Exec, Graphics and Intuition libraries don’t need to be opened and their function offsets are built in to E. That’s why you won’t find, for example, a dos.m file in Emodules:. The constants and objects for these libraries still need to be included via modules (they are not built in to E).

1.122 beginner.guide/Non-Standard Modules

```
Non-Standard Modules
====================
```

Several non-standard library modules are also supplied with Amiga E. To make your own modules you need the pragma2module and iconvert programs. These convert standard format C header files and Assembly include files to modules. The C header file should contain pragmas for function offsets, and the Assembly include file should contain constant and structure definitions (the Assembly structures will be converted to objects). However, unless you're trying to do really advanced things you probably don't need to worry about any of this!

1.123 beginner.guide/Example Module Use

```
                Example Module Use
==================
```

The gadget example program in Part Three shows how to use constants from the module intuition/intuition (see Gadgets), and the IDCMP example program shows the object gadget from that module being used (see

IDCMP Messages). The following program uses the modules for the Reqtools library, which is not a standard Amiga system library but a commonly used one, and the appropriate modules are supplied with Amiga E. To run this program, you will, of course, need the reqtools.library in Libs:.

```
     MODULE 'reqtools'

     PROC main()
       DEF col
       IF (reqtoolsbase:=OpenLibrary('reqtools.library',37))<>NIL
         IF (col:=RtPaletteRequestA('Select a colour', 0,0))<>-1
           RtEZRequestA('You picked colour \d',
                        'I did|I can\at remember',0,[col],0)
         ENDIF
         CloseLibrary(reqtoolsbase)
       ELSE
         WriteF('Could not open reqtools.library, version 37+\n')
       ENDIF
     ENDPROC
```

The reqtoolsbase variable is the library base variable for the Reqtools library. This is defined in the module reqtools and you must store the result of the OpenLibrary call in this variable if you are going to use any of the functions from the Reqtools library. (You can find out which variable to use for other libraries by running the showmodule program on the library module for the library.) The two functions the program uses are RtPaletteRequestA and RtEZRequestA. Without the inclusion of the reqtools module and the setting up of the reqtoolsbase variable you would not be able to use these functions. In fact, if you didn’t have the MODULE line you wouldn’t even be able to compile the program because the compiler wouldn’t know where the functions came from and would complain bitterly.

Notice that the Reqtools library is closed before the program terminates (if it had been successfully opened). This is always necessary: if you succeed in opening a library you must close it when you’re finished with it.

1.124 beginner.guide/Code Modules

Code Modules
============

You can also make modules containing procedure definitions and some global variables. These are called code modules and can be extremely useful. This section briefly outlines their construction and use. For in-depth details see the ‘Reference Manual’.

Code modules can be made by using the E compiler as you would to make an executable, except you put the statement OPT MODULE at the start of the code. Also, all definitions that are to be accessed from outside the module need to be marked with the EXPORT keyword. Alternatively, all definitions can be exported using OPT EXPORT at the start of the code. You include the definitions from this module (and use the exported ones) in your program using MODULE in the normal way.

The following code is an example of a small module:

```
OPT MODULE

EXPORT CONST MAX_LEN=20

EXPORT OBJECT fullname
  firstname, surname
ENDOBJECT

EXPORT PROC printname(p:PTR TO fullname)
  IF short(p.surname)
    WriteF(’Hello, \s \s\n’, p.firstname, p.surname)
  ELSE
    WriteF(’Gosh, you have a long name\n’)
  ENDIF
ENDPROC

PROC short(s)
  RETURN StrLen(s)<MAX_LEN
ENDPROC
```

Everything is exported except the short procedure. Therefore, this can be accessed only in the module. In fact, the printname procedure uses it (rather artificially) to check the length of the surname. It's not of much use or interest apart from in the module, so that's why it isn't exported. In effect, we've hidden the fact that printname uses short from the user of the module.

Assuming the above code was compiled to module mymods/name, here's how it could be used:

```
MODULE 'mymods/name'

PROC main()
  DEF fred:PTR TO fullname, bigname
  fred.firstname:='Fred'
  fred.surname:='Flintstone'
  printname(fred)
  bigname:=['Peter', 'Extremelybiglongprehistoricname']
  printname(bigname)
ENDPROC
```

Global variables in a module are a bit more problematic than the other kinds of definitions. You cannot initialise them in the declaration or make them reserve chunks memory. So you can't have ARRAY, OBJECT, STRING or LIST declarations. However, you can have pointers so this isn't a big problem. The reason for this limitation is that exported global variables with the same name in a module and the main program are taken to be the same variable, and the values are shared. So you can have an array declaration in the main program:

```
DEF a[80]:ARRAY OF INT
```

and the appropriate pointer declaration in the module:

```
EXPORT DEF a:PTR TO INT
```

The array from the main program can then be accessed in the module! For this reason you also need to be pretty careful about the names of your exported variables so you don't get unwanted sharing. Global variables which are not exported are private to the module, so will not clash with variables in the main program or other modules.

1.125 beginner.guide/Exception Handling

Exception Handling

****

Often your program has to check the results of functions and do different things if errors have occurred. For instance, if you try to open a window (using OpenW), you may get a NIL pointer returned which shows that the window could not be opened for some reason. In this case you normally can't continue with the program, so you must tidy up and terminate. Tidying up can sometimes involve closing windows, screens and libraries, so sometimes your error cases can make your program cluttered and messy. This is where exceptions come in--an exception is simply an error case, and exception handling is dealing with error cases. The exception handling in E neatly separates error specific code from the real code of your program.

1.126 beginner.guide/Procedures with Exception Handlers

Procedures with Exception Handlers
==================================

A procedure with an exception handler looks like this:

```
PROC fred(params...) HANDLE
  /* Main, real code */
EXCEPT
  /* Error handling code */
ENDPROC
```

This is very similar to a normal procedure, apart from the HANDLE and EXCEPT keywords. The HANDLE keyword means the procedure is going to have an exception handler, and the EXCEPT keyword marks the end of the normal code and the start of the exception handling code. The procedure works just as normal, executing the code in the part before the EXCEPT, but when an error happens you can pass control to the exception handler (i.e., the code after the EXCEPT is executed).

1.127 beginner.guide/Raising an Exception

Raising an Exception
====================

When an error occurs (and you want to handle it), you raise an exception using either the Raise or Throw function. You call Raise with a number which identifies the kind of error that occurred. The code in the exception handler is responsible for decoding the number and then doing the appropriate thing. Throw is very similar to Raise, and the following description of Raise also applies to Throw. The difference is that Throw takes a second argument which can be used to pass extra information to a handler (usually a string). The terms ‘raising’ and ‘throwing’ an exception can be used interchangeably.

When Raise is called it immediately stops the execution of the current procedure code and passes control to the exception handler of most recent procedure which has a handler (which may be the current procedure). This is a bit complicated, but you can stick to raising exceptions and handling them in the same procedure, as in the next example:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Raise(ERR_MEM)
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF('Not enough memory\n')
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
ENDPROC
```

This uses an exception handler to print a message saying there wasn't enough memory if the call to New returns NIL. The parameter to Raise is stored in the special variable exception in the exception handler part of the code, so if Raise is called with a number other than ERR_MEM a message saying "Unknown exception" will be printed.

Try running this program with a really large BIG_AMOUNT constant, so that the New can't allocate the memory. Notice that the "Got enough memory" is not printed if Raise is called. That's because the execution of the normal procedure code stops when Raise is called, and control passes to the appropriate exception handler. When the end of the exception handler is reached the procedure is finished, and in this case the program terminates because the procedure was the main procedure.

If Throw is used instead of Raise then, in the handler, the special variable exceptioninfo will contain the value of the second parameter. This can be used in conjunction with exception to provide the handler with more information about the error. Here's the above example re-written to use Throw:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Throw(ERR_MEM, 'Not enough memory\n')
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF(exceptioninfo)
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
```

```
ENDPROC
```

An enumeration (using ENUM) is a good way of getting different constants for various exceptions. It's always a good idea to use constants for the parameter to Raise and in the exception handler, because it makes everything a lot more readable: Raise(ERR_MEM) is much clearer than Raise(1). The enumeration starts at one because zero is a special exception: it usually means that no error occurred. This is useful when the handler does the same cleaning up that would normally be done when the program terminates successfully. For this reason there is a special form of EXCEPT which automatically raises a zero exception when the code in the procedure successfully terminates. This is EXCEPT DO, with the DO suggesting to the reader that the exception handler is called even if no error occurs. Also, the argument to the Raise function defaults to zero (see

Default Arguments

) if it is omitted.

So, what happens if you call Raise in a procedure without an exception handler? Well, this is where the real power of the handling mechanism comes to light. In this case, control passes to the exception handler of the most recent procedure with a handler. If none are found then the program terminates. Recent means one of the procedures involved in calling your procedure. So, if the procedure fred calls barney, then when barney is being executed fred is a recent procedure. Because the main procedure is where the program starts it is a recent procedure for every other procedure in the program. This means, in practice:

* If you define fred to be a procedure with an exception handler then any procedures called by fred will have their exceptions handled by the handler in fred if they don't have their own handler.

* If you define main to be a procedure with an exception handler then any exceptions that are raised will always be dealt with by some exception handling code (i.e., the handler of main or some other procedure).

Here's a more complicated example:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Hello from main\n')
  fred()
  barney()
  WriteF('Goodbye from main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Hello from fred\n')
  Raise(FRED)
  WriteF(' Goodbye from fred\n')
EXCEPT
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
```

```
    WriteF('  Hello from barney\n')
    Raise(BARNEY)
    WriteF('  Goodbye from barney\n')
  ENDPROC
```

When you run this program you get the following output:

```
Hello from main
 Hello from fred
 Handler fred: 1
  Hello from barney
```

This is because the fred procedure is terminated by the Raise(FRED) call, and the whole program is terminated by the Raise(BARNEY) call (since barney and main do not have handlers).

Now try this:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Hello from main\n')
  fred()
  WriteF('Goodbye from main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Hello from fred\n')
  barney()
  Raise(FRED)
  WriteF(' Goodbye from fred\n')
EXCEPT
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
  WriteF('  Hello from barney\n')
  Raise(BARNEY)
  WriteF('  Goodbye from barney\n')
ENDPROC
```

When you run this you get the following output:

```
Hello from main
 Hello from fred
  Hello from barney
 Handler fred: 2
Goodbye from main
```

Now the fred procedure calls barney, so main and fred are recent procedures when Raise(BARNEY) is executed, and therefore the fred exception handler is called. When this handler finishes the call to fred in main is finished, so the main procedure is completed and we see the 'Goodbye' message. In the previous program the Raise(BARNEY) call did not get handled and the whole program terminated at that point.

1.128 beginner.guide/Automatic Exceptions

Automatic Exceptions
====================

In the previous section we saw an example of raising an exception when a call to New returned NIL. We can re-write this example to use automatic exception raising:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

RAISE ERR_MEM IF New()=NIL

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF('Not enough memory\n')
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
ENDPROC
```

The only difference is the removal of the IF which checked the value of block, and the addition of a RAISE part. This RAISE part means that whenever the New function is called in the program, the exception ERR_MEM will be raised if it returns NIL (i.e., the exception ERR_MEM is automatically raised). This unclutters the program by removing a lot of error checking IF statements.

The precise form of the RAISE part is:

```
RAISE exception  IF function()  compare  value ,
      exception2 IF function2() compare2 value2 ,
      ...
```

The exception is a constant (or number) which represents the exception to be raised, function is the E built-in or system function to be automatically checked, value is the return value to be checked against, and compare is the method of checking (i.e., =, <>, <, <=, > or >=). This mechanism only exists for built-in or library functions because they would otherwise have no way of raising exceptions. The procedures you define yourself can, of course, use Raise to raise exceptions in a much more flexible way.

1.129 beginner.guide/Raise within an Exception Handler

```
                Raise within an Exception Handler
=================================
```

If you call Raise within an exception handler then control passes to the next most recent handler. In this way you can write procedures which have handlers that perform local tidying up. By using Raise at the end of the handler code you can invoke the next layer of tidying up.

As an example we'll use the Amiga system functions AllocMem and FreeMem which are like the built-in function New and Dispose, but the memory allocated by AllocMem must be deallocated (using FreeMem) when it's finished with, before the end of the program.

```
     CONST SMALL=100, BIG=123456789

     ENUM ERR_MEM=1

     RAISE ERR_MEM IF AllocMem()=NIL

     PROC main()
       allocate()
     ENDPROC

     PROC allocate() HANDLE
       DEF mem=NIL
       mem:=AllocMem(SMALL, 0)
       morealloc()
       FreeMem(mem, SMALL)
     EXCEPT
       IF mem THEN FreeMem(mem, SMALL)
       WriteF('Handler: deallocating "allocate" local memory\n')
     ENDPROC

     PROC morealloc() HANDLE
       DEF more=NIL, andmore=NIL
       more:=AllocMem(SMALL, 0)
       andmore:=AllocMem(BIG, 0)
       WriteF('Allocated all the memory!\n')
       FreeMem(andmore, BIG)
       FreeMem(more, SMALL)
     EXCEPT
       IF andmore THEN FreeMem(andmore, BIG)
       IF more THEN FreeMem(more, SMALL)
       WriteF('Handler: deallocating "morealloc" local memory\n')
       Raise(ERR_MEM)
     ENDPROC
```

The calls to AllocMem are automatically checked, and if NIL is returned the exception ERR_MEM is raised. The handler in the allocate procedure checks to see if it needs to free the memory pointed to by mem, and the handler in the morealloc checks andmore and more. At the end of the morealloc handler is the call Raise(ERR_MEM). This passes control to the exception handler of the allocate procedure, since allocate called morealloc.

There's a couple of subtle points to notice about this example. Firstly, the memory variables are all initialised to NIL. This is because the automatic exception raising on AllocMem will result in the variables not being assigned if the call returns NIL (i.e., the exception is raised before the assignment takes place), and the handler needs them to be NIL if AllocMem fails. Of course, if AllocMem does not return NIL the assignments work as normal.

Secondly, the IF statements in the handlers check the memory pointer variables do not contain NIL by using their values as truth values. Since NIL is actually zero, a non-NIL pointer will be non-zero, i.e., true in the IF check. This shorthand is often used, and so you should be aware of it.

It is quite common that an exception handler will want to raise the same exception after it has done its processing. The function ReThrow (which has no arguments) can be used for this purpose. It will re-raise the exception, but only if the exception is not zero (since this special value means that no error occurred). If the exception is zero then this function has no effect. In fact, the following code fragments (within a handler) are equivalent:

```
       ReThrow()

       IF exception THEN Throw(exception, exceptioninfo)
```

There are two examples, in Part Three, of how to use an exception handler to make a program more readable: one deals with using data files (see
String Handling and I-O
) and the other deals with opening screens and windows (see
Screens
).

1.130 beginner.guide/Memory Allocation

Memory Allocation
****

When a program is running memory is being used in various different ways. In order to use any memory it must first be allocated, which is simply a way of marking memory as being ‘in use’. This is to prevent the same piece of memory being used for different data storage (e.g., by different programs), and so helps prevent corruption of the data stored there. There are two general ways in which memory can be allocated: dynamically and statically.

Static Allocation

Deallocation of Memory

Dynamic Allocation

NEW and END Operators

1.131 beginner.guide/Static Allocation

Static Allocation
=================

Statically allocated memory is memory allocated by the program for variables and static data like string constants, lists and typed lists (see

Static data

). Every variable in a program requires some memory in which to store its value. Variables declared to be of type ARRAY, LIST, STRING or any object require two lots of memory: one to hold the value of the pointer and one to hold the large amount of data (e.g., the elements in an ARRAY). In fact, such declarations are merely PTR TO type declarations together with an initialisation of the pointer to the address of some (statically) allocated memory to hold the data. The following example shows very similar declarations, with the difference being that in the second case (using PTR) only memory to hold the pointer values is allocated. The first case also allocates memory to hold the appropriate size of array, object and E-string.

```
       DEF a[20]:ARRAY,   m:myobj,        s[10]:STRING

       DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
```

The pointers in the second case are not initialised by the declaration and, therefore, they are not valid pointers. This means that they should not be dereferenced in any way, until they have been initialised to the address of some allocated memory. This usually involves dynamic allocation of memory (see

Dynamic Allocation

).

1.132 beginner.guide/Deallocation of Memory

Deallocation of Memory
======================

When memory is allocated it is, conceptually, marked as being ‘in use’. This means that this piece of memory cannot be allocated again, so a different piece will be allocated (if any is available) when the program wants to allocate some more. In this way, variables are allocated different pieces of memory, and so their values can be distinct. But there is only a certain amount of memory available, and if it could not be marked as ‘not in use’ again it would soon run out (and the program would come to a nasty end). This is what deallocation does: it marks previously allocated memory as being ‘not in use’ and so makes it available for allocation again. However, memory should be deallocated only when it is actually no longer in use, and this is where things get a bit complicated.

Memory is such a vital resource in every computer that it is important to use as little of it as necessary and to deallocate it whenever possible. This is why a programming language like E handles most of the memory allocation for variables. The memory allocated for variables can be automatically deallocated when it is no longer possible for the program to use that variable. However, this automatic deallocation is not useful for global variables, since they can be used from any procedure and so can be deallocated only when the program terminates. A procedure’s local variables, on the other hand, are allocated when the procedure is called but cannot be used after the procedure returns. They can, therefore, be deallocated when the procedure returns.

Pointers, as always, can cause big problems. The following example shows why you need to be careful when using pointers as the return value of a procedure.

```
/* This is an example of what *NOT* to do */
PROC fullname(first, last)
  DEF full[40]:STRING
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

PROC main()
  WriteF('Name is \s\n', fullname('Fred', 'Flintstone'))
ENDPROC
```

On first sight this seems fine, and, in fact, it may even work correctly if you run it once or twice (but be careful: it could crash your machine). The problem is that the procedure fullname returns the value of the local variable full, which is a pointer to some statically allocated memory for the E-string and this memory will be deallocated when the procedure returns. This means that the return value of any call to fullname is the address of recently deallocated memory, so it is invalid to dereference it. But the call to WriteF does just that: it dereferences the result of fullname in order to print the E-string it points to. This is a very common problem, because it is such an easy thing to do. The fact that it may, on many occasions, appear to work makes it much harder to find, too. The solution, in this case, is to use dynamic allocation (see

Dynamic Allocation
).

If you’re still a bit sceptical that this really is a problem, try the above fullname procedure definition with either of these replacement main procedures, but be aware, again, that each one has the potential to crash your machine.

```
/* This might not print the correct string */
PROC main()
  DEF f
  f:=fullname('Fred', 'Flintstone')
  WriteF('Name is \s\n', f)
ENDPROC

/* This will definitely print g instead of f */
PROC main()
  DEF f, g
  f:=fullname('Fred', 'Flintstone')
  g:=fullname('Barney', 'Rubble')
  WriteF('Name is \s\n', f)
ENDPROC
```

(The reason why things go wrong is outlined above, but the reasons why each prints what it does is beyond the scope of this Guide.)

1.133 beginner.guide/Dynamic Allocation

Dynamic Allocation
==================

Dynamically allocated memory is any memory that is not statically allocated. To allocate memory dynamically you can use the List and String functions, all flavours of New, and the versatile NEW operator. But because the memory is dynamically allocated it must be explicitly deallocated when no longer needed. In all the above cases, though, any memory that is still allocated when the program terminates will be deallocated automatically.

Another way to allocate memory dynamically is to use the Amiga system functions based on AllocMem. However, these functions require that the memory allocated using them be deallocated (using functions like FreeMem) before the program terminates, or else it will never be deallocated (not until your machine is rebooted, anyway). It is safer, therefore, to try to use the E functions for dynamic allocation whenever possible.

There are many reasons why you might want to use dynamic allocation, and most of them involve initialisation of pointers. For example, the declarations in the section about static allocation can be extended to give initialisations for the pointers declared in the second DEF line (see

Static Allocation
).

```
  DEF a[20]:ARRAY,   m:myobj,        s[10]:STRING

  DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
  a:=New(20)
  m:=New(SIZEOF myobj)
  s:=String(20)
```

These are initialisations to dynamically allocated memory, whereas the first line of declarations initialise similar pointers to statically allocated memory. If these sections of code were part of a procedure then, since they would now be local variables, there would be one other, significant difference: the dynamically allocated memory would not automatically be deallocated when the procedure returns, whereas the statically allocated memory would. This means that we can solve the deallocation problem (see
Deallocation of Memory
).

```
/* This is the correct way of doing it */
PROC fullname(first, last)
  DEF full
  full:=String(40)
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

PROC main()
  DEF f, g
  WriteF('Name is \s\n', fullname('Fred', 'Flintstone'))
  f:=fullname('Fred', 'Flintstone')
  g:=fullname('Barney', 'Rubble')
  WriteF('Name is \s\n', f)
ENDPROC
```

The memory for the E-string pointed to by full is now allocated dynamically, using String, and is not deallocated until the end of the program. This means that it is quite valid to pass the value of full as the result of the procedure fullname, and it is quite valid to dereference the result by printing it using WriteF. However, this has caused one last problem: the memory is not deallocated until the end of the program, so is potentially wasted since it could be used, for example, to hold the results of subsequent calls. Of course, the memory can be deallocated only when the data it stores is no longer required. The following replacement main procedure shows when you might want to deallocate the E-string (using DisposeLink).

```
PROC main()
  DEF f, g
  f:=fullname('Fred', 'Flintstone')
  WriteF('Name is \s, f points to $\h\n', f, f)
/* Try this with and without the next DisposeLink line */
  DisposeLink(f)
  g:=fullname('Barney', 'Rubble')
  WriteF('Name is \s, g points to $\h\n', g, g)
  DisposeLink(g)
ENDPROC
```

If you run this with the DisposeLink(f) line you'll probably find that g will be a pointer to the same memory as f. This is because the call to DisposeLink has deallocated the memory pointed to by f, so it can be reused to store the E-string pointed to by g. If you comment out (or delete) the DisposeLink line, then you will find that f and g always point to different memory.

In some ways it is best to never do any deallocation, because of the problems you can get into if you deallocate memory too early (i.e., before you've finished with the data it contains). Of course, it is safe (but temporarily wasteful) to do this with the E dynamic allocation functions, but it is very wasteful (and wrong) to do this with the Amiga system functions like AllocMem.

Another benefit of using dynamic allocation is that the size of the arrays, E-lists and E-strings that can be created can be the result of any expression, so is not restricted to constant values. (Remember that the size given on ARRAY, LIST and STRING declarations must be a constant.) This means that the fullname procedure can be made more efficient and allocate only the amount of memory it needs for the E-string it creates.

```
PROC fullname(first, last)
  DEF full
  /* The extra +1 is for the added space */
  full:=String(StrLen(first)+StrLen(last)+1)
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full
```

However, it may be very complicated or inefficient to calculate the correct size. In these cases, a quick, constant estimate might be better, overall.

The various functions for allocating memory dynamically have corresponding functions for deallocating that memory. The following table shows some of the more common pairings.

Allocation	Deallocation
New	Dispose
NewR	Dispose
List	DisposeLink
String	DisposeLink
NEW	END
FastNew	FastDispose
AllocMem	FreeMem
AllocVec	FreeVec
AllocDosObject	FreeDosObject

NEW and END are versatile and powerful operators, discussed in the following section. The functions beginning with Alloc- are Amiga system functions and are paired with similarly suffixed functions with a Free-prefix. See the 'Rom Kernel Reference Manual' for more details.

1.134 beginner.guide/NEW and END Operators

```
                NEW and END Operators
=====================
```

To help deal with dynamic allocation and deallocation of memory there are two, powerful operators, NEW and END. The NEW operator is very versatile, and similar in operation to the New family of built-in functions (see System support functions). The END operator is the deallocating complement of NEW (so it is similar to the Dispose family of built-in functions). The major difference between NEW and the various flavours of New is that NEW allocates memory based on the types of its arguments.

Object and simple typed allocation

Array allocation

List and typed list allocation

OOP object allocation

1.135 beginner.guide/Object and simple typed allocation

Object and simple typed allocation

The following sections of code are roughly equivalent and serve to show the function of NEW, and how it is closely related to NewR. (The type can be any object or simple type.)

```
DEF p:PTR TO type
NEW p

DEF p:PTR TO type
p:=NewR(SIZEOF type)
```

Notice that the use of NEW is not like a function call, as there are no parentheses around the parameter p. This is because NEW is an operator rather than a function. It works differently from a function, since it also needs to know the types of its arguments. This means that the declaration of p is very important, since it governs how much memory is allocated by NEW. The version using NewR explicitly gives the amount of memory to be allocated (using the SIZEOF operator), so in this case the declared type of p is not so important for correct allocation.

The next example shows how NEW can be used to initialise several pointers at once. The second section of code is roughly equivalent, but uses NewR. (Remember that the default type of a variable is LONG, which is actually PTR TO CHAR.)

```
DEF p:PTR TO LONG, q:PTR TO myobj, r
NEW p, q, r
```

```
DEF p:PTR TO LONG, q:PTR TO myobj, r
p:=NewR(SIZEOF LONG)
q:=NewR(SIZEOF myobj)
r:=NewR(SIZEOF CHAR)
```

These first two examples have shown the statement form of NEW. There is also an expression form, which has one parameter and returns the address of the newly allocated memory as well as initialising the argument pointer to this address.

```
DEF p:PTR TO myobj, q:PTR TO myobj
q:=NEW p

DEF p:PTR TO myobj, q:PTR TO myobj
q:=(p:=NewR(SIZEOF type))
```

This may not seem desperately useful, but it's also the way that NEW is used to allocate copies of lists and typed lists (see

List and typed list allocation
).

To deallocate memory allocated using NEW you use the END statement with the pointers that you want to deallocate. To work properly, END requires that the type of each pointer matches the type used when it was allocated with NEW. Failure to do this will result in an incorrect amount of memory being deallocated, and this can cause many subtle problems in a program. You must also be careful not to deallocate the same memory twice, and to this end the pointers given to END are re-initialised to NIL after the memory they point to is deallocated (it is quite safe to use END with a pointer which is NIL). This does not catch all problems, however, since more than one pointer can point to the same piece of memory, as shown in the example below.

```
DEF p:PTR TO LONG, q:PTR TO LONG
q:=NEW p
p[]:=-24
q[]:=613
END p
/* p is now NIL, but q is now invalid but not NIL */
```

The first assignment initialises q to be the same as p (which is initialised by NEW). Both the next two assignments change the value pointed to by both p and q. The memory allocated to store this value is then deallocated, using END, and this also sets p to NIL. However, the address stored in q is not altered, and still points to the memory that has just been deallocated. This means that q now has a plausible, but invalid, pointer value. The only thing that can safely be done with q is re-initialise it. One of the worst things that could be done is to use it with END, which would deallocate the same memory again, and potentially crash your machine. So, in summary, don't deallocate the same pointer value more than once, and keep track of which variables point to the same memory as others.

Just as a use of NEW has a simple (but rough) equivalent using NewR, END has an equivalent using Dispose, as shown by the following sections of code.

```
       END p

       IF p
         Dispose(p)
         p:=NIL
       ENDIF
```

In fact, it's a tiny bit more complicated than that, since OOP objects are allocated and deallocated using NEW and END (see Object Oriented E).

1.136 beginner.guide/Array allocation

Array allocation

Arrays can also be allocated using NEW, and this works in a very similar way to that outlined in the previous section. The difference is that the size of the array must also be supplied, in both the use of NEW and END. Of course, the size supplied to END must be the same as the size supplied to the appropriate use of NEW. All this extra effort also gains you the ability to create an array of a size which is not a constant (unlike variables of type ARRAY). This means that the size supplied to NEW and END can be the result of an arbitrary expression.

```
       DEF a:PTR TO LONG, b:PTR TO myobj, s
       NEW a[10]  /* A dynamic array of LONG  */
       s:=my_random(20)
       NEW b[s]   /* A dynamic array of myobj */
       /* ...some other code... */
       END a[10], b[s]
```

The my_random function stands for some arbitrary calculation, to show that s does not have to be a constant. This form of NEW can also be used as an expression, as before.

1.137 beginner.guide/List and typed list allocation

List and typed list allocation

Lists and typed lists are usually static data, but NEW can be used to create dynamically allocated versions. This form of NEW can be used only as an expression, and it takes the list (or typed list) as its argument and returns the address of the dynamically allocated copy of the list. Deallocation of the memory allocated in this way is a bit more complicated than before, but you can, of course, let it be deallocated automatically at the end of the program.

The following example shows how simple it is to use NEW to cure the static data problem described previously (see Static data). The difference from the original, incorrect program is very subtle.

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=NEW [1, i, i*i]
      /* a[i] is now dynamically allocated */
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    WriteF('a[\d] is an array at address \d\n', i, p)
    WriteF('  and the second element is \d\n', p[1])
  ENDFOR
ENDPROC
```

The minor alteration is to prefix the list with NEW, thereby making the list dynamic. This means that each a[i] is now a different list, rather than the same, static list of the original version of the program.

Typed lists are allocated in a similar way, and the following example also shows how to deallocate this memory. Basically, you need to know how long the new array is (i.e., how many elements there are), since a typed list is really just an initialised array. You can then deallocate it like a normal array, remembering to use an appropriately typed pointer. Object-typed lists are restricted (when used with NEW) to an array of at most one object, so is useful only for allocating an initialised object (not really an array). Notice how, in the following code, the pointer q can be treated both as an object and as an array of one object (see

Element selection and element types).

```
OBJECT myobj
  x:INT, y:LONG, z:INT
ENDOBJECT

PROC main()
  DEF p:PTR TO INT, q:PTR TO myobj
  p:=NEW [1, 9, 3, 7, 6]:INT
  q:=NEW [1, 2]:myobj
  WriteF('Last element in array p is \d\n', p[4])
  WriteF('Object q is x=\d, y=\d, z=\d\n',
         q.x,   q.y,   q.z)
  WriteF('Array q is q[0].x=\d, q[0].y=\d, q[0].z=\d\n',
         q[].x, q[].y, q[].z)
  END p[5], q
ENDPROC
```

The dynamically allocated version of an object-typed list differs from the static version in another way: it always has memory allocated for a whole number of objects, so a partially initialised object is padded with zero elements. The static version does not allocate this extra padding, so you must be careful not to access any element beyond those mentioned in the list.

The deallocation of NEW copies of normal lists can, as ever, be left to be done automatically at the end of the program. If you want to deallocate them before this time you must use the function FastDisposeList, passing the address of the list as the only argument. You must not use END or any other method of deallocation. FastDisposeList is the only safe way of deallocating lists allocated using NEW.

1.138 beginner.guide/OOP object allocation

OOP object allocation

Currently, the only way to create OOP objects in E is to use NEW and the only safe way to destroy them is to use END. This is probably the most common use of NEW and END and is described in detail later (see

Objects in E
).

1.139 beginner.guide/Floating-Point Numbers

Floating-Point Numbers
****

Floating-point or real numbers can be used to represent both very small fractions and very large numbers. However, unlike a LONG which can hold every integer in a certain range (see
Variable types
), floating-point
numbers have limited accuracy. Be warned, though: using floating-point arithmetic in E is quite complicated and most problems can be solved without using floating-point numbers, so you may wish to skip this chapter until you really need to use them.

Floating-Point Values

Floating-Point Calculations

Floating-Point Functions

Accuracy and Range

1.140 beginner.guide/Floating-Point Values

Floating-Point Values
=====================

Floating-point values in E are written just like you might expect and are stored in LONG variables:

```
DEF x
x:=3.75
x:=-0.0000367
x:=275.0
```

You must remember to use a decimal point (without any spaces around it) in the number if you want it to be considered a floating-point number, and this is why a trailing .0 was used on the number in the last assignment. At present you can't express every floating-point value in this way; the compiler may complain that the value does not fit in 32-bits if you try to use more than about nine digits in a single number. You can, however, use the various floating-point maths functions to calculate any value you want (see
Floating-Point Functions
).

1.141 beginner.guide/Floating-Point Calculations

Floating-Point Calculations
===========================

Since a floating-point number is stored in a LONG variable it would normally be interpreted as an integer, and this interpretation will generally not give a number anything like the intended floating-point number. To use floating-point numbers in expressions you must use the (rather complicated) floating-point conversion operator, which is the ! character. This converts expressions and the normal maths and comparison operators to and from floating-point.

All expressions are, by default, integer expressions. That is, they represent LONG integer values, rather than floating-point values. The first time a ! occurs in an expression the value of the expression so far is converted to floating-point and all the operators and variables after this point are considered floating-point. The next time it occurs the (floating-point) value of the expression so far is converted to an integer, and the following operators and variables are considered integer again. You can use ! as often as necessary within an expression. Parts of an expression in parentheses are treated as separate expressions, so are, by default, integer expressions (this, includes function call arguments).

The integer/floating-point conversions performed by ! are not simple. They involve rounding and also bounding. Conversion, for example, from integer to floating-point and back again will generally not result in the original integer value.

Here’s a few commented examples, where f always holds a floating-point number, and i and j always hold integers:

```
DEF f, i, j
i:=1
f:=1.0
f:=i!  -> i converted to floating-point (1.0)
f:=6.2
i:=!f! -> the expression f is floating-point,
       -> then converted to integer (6)
```

In the first assignment, the integer value one is assigned to i. In the second, the floating-point value 1.0 is assigned to f. The expression on the right-hand side of third assignment is considered to be an integer until the ! is met, at which point it is converted to the nearest floating-point value. So, f is assigned the floating-point value of one (i.e., 1.0), just like it is by the second assignment. The expression in the final assignment needs to start off as floating-point in order to interpret the value stored in f as floating-point. The expression finishes by converting back to integer. The overall result is to turn the floating-point value of f into the nearest integer (in this case, six).

The assignments below are more complicated, but should be straight-forward to follow. Again, f always holds a floating-point number, and i and j always hold integers.

```
f:=!f*f -> the whole expression is floating-point,
        -> and f is squared (6.2*6.2)
f:=!f*(i!) -> the whole expression is floating-point,
           -> i is converted to floating-point and
           -> multiplied by f
j:=!f/(i!)! -> the whole division is floating-point,
            -> with the result converted to integer
j:=!f!/i -> floating-point f is converted to integer
         -> and is (integer) divided by i
IF !f<230.0 THEN RETURN 0  -> floating-point comparison <
IF !f>(i!)  THEN RETURN 0  -> i converted to floating-point,
                           -> then compared to f
```

If the ! were omitted from the first assignment, then not only would the value in f be interpreted (incorrectly) as integer, but the multiplication performed would be integer multiplication, rather than floating-point. In the second assignment, the parentheses around the expression involving i are crucial. Without the parentheses the value stored in i would be interpreted as floating-point. This would be wrong because i actually stores an integer value, so parentheses are used to start a new expression (which defaults to being integer). The value of i is then interpreted correctly, and finally converted to floating-point (by the ! just before the closing parenthesis). The (floating-point) multiplication then takes place with two floating-point values, and the result is stored in f. In the last two assignments (using division), j is assigned roughly the same value. However, the expression in the first assignment allows for greater accuracy, since it uses floating-point division. This means the result will be rounded, whereas it is truncated when integer division is used.

One important thing to know about floating-point numbers in E is that the following assignments store the same value in g (again, f stores a floating-point number). This is because no computation is performed and no conversion happens: the value in f is simply copied to g. This is especially important for function calls, as we shall see in the next section. Strictly speaking, however, the second version is better, since it shows (to the reader of the code) that the value in f is meant to be floating-point.

```
       g:=f
       g:=!f
```

1.142 beginner.guide/Floating-Point Functions

Floating-Point Functions
========================

There are functions for formatting floating-point numbers to E-strings (so that they can be printed) and for decoding floating-point numbers from strings. There are also a number of built-in, floating-point functions which compute some of the less common mathematical functions, such as the various trigonometric functions.

RealVal(string)

This works in a similar way to Val for extracting integers from a string. The decoded floating-point value is returned as the regular return value, and the number of characters of string that were read to make the number is returned as the first optional return value. If a floating-point value could not be decoded from the string then zero is returned as the optional return value and the regular return value will be zero (i.e., 0.0).

RealF(e-string,float,digits)

Converts the floating-point value float into a string which is stored in e-string. The number of digits to use after the decimal point is specified by digits, which can be zero to eight. The floating-point value is rounded to the specified number of digits. A value of zero for digits gives a result with no fractional part and no decimal point. The e-string is returned by this function, and this makes it easy to use with WriteF.

```
          PROC main()
            DEF s[20]:STRING, f, i
            f:=21.60539
            FOR i:=0 TO 8
              WriteF('f is \s (using digits=\d)\n', RealF(s, f, i), i)
            ENDFOR
          ENDPROC
```

Notice that the floating-point argument, f, to RealF does not need a leading ! because we are simply passing its value and not performing a computation with it. The program should generate the following output:

```
          f is 22 (using digits=0)
          f is 21.6 (using digits=1)
          f is 21.61 (using digits=2)
          f is 21.605 (using digits=3)
          f is 21.6054 (using digits=4)
          f is 21.60539 (using digits=5)
          f is 21.605390 (using digits=6)
          f is 21.6053900 (using digits=7)
          f is 21.60539000 (using digits=8)
```

`Fsin(float), Fcos(float), Ftan(float)`

These compute the sine, cosine and tangent (respectively) of the supplied float angle, which is specified in radians.

`Fabs(float)`

Returns the absolute value of float, much like Abs does for integers.

`Ffloor(float), Fceil(float)`

The Ffloor function rounds a floating-point value down to the nearest, whole floating-point value. The Fceil function rounds it up.

`Fsqrt(float)`

Returns the square root of float.

`Fpow(x,y), Fexp(float)`

The Fpow function returns the value of x raised to the power of y (which are both floating-point values). The Fexp function returns the value of e raised to the power of float, where e is the mathematically special value (roughly 2.718282). ‘Raising to a power’ is known as exponentiation.

`Flog10(float), Flog(float)`

The Flog10 function returns the log to base ten of float (the common logarithm). The Flog function returns the log to base e of float (the natural logarithm). Flog10 and Fpow are linked in the following way (ignoring floating-point inaccuracies):

```
       x = Fpow(10.0, Flog10(x))
```

Flog and Fexp are similarly related (Fexp could be used again, using 2.718282 as the first argument in place of 10.0).

```
       x = Fexp(Flog(x))
```

Here’s a small program which uses a few of the above functions, and shows how to define functions which use and/or return floating-point values.

```
DEF f, i, s[20]:STRING

PROC print_float()
  WriteF(’\tf is \s\n’, RealF(s, !f, 8))
ENDPROC

PROC print_both()
  WriteF(’\ti is \d, ’, i)
```

```
  print_float()
ENDPROC

/* Square a float */
PROC square_float(f) IS !f*f

/* Square an integer */
PROC square_integer(i) IS i*i

/* Converts a float to an integer */
PROC convert_to_integer(f) IS Val(RealF(s, !f, 0))

/* Converts an integer to a float */
PROC convert_to_float(i) IS RealVal(StringF(s, '\d', i))

/* This should be the same as Ftan */
PROC my_tan(f) IS !Fsin(!f)/Fcos(!f)

/* This should show float inaccuracies */
PROC inaccurate(f) IS Fexp(Flog(!f))

PROC main()
  WriteF('Next 2 lines should be the same\n')
  f:=2.7; i:=!f!
  print_both()
  f:=2.7; i:=convert_to_integer(!f)
  print_both()

  WriteF('Next 2 lines should be the same\n')
  i:=10;  f:=i!
  print_both()
  i:=10;  f:=convert_to_float(i)
  print_both()

  WriteF('f and i should be the same\n')
  i:=square_integer(i)
  f:=square_float(f)
  print_both()

  WriteF('Next 2 lines should be the same\n')
  f:=Ftan(.8)
  print_float()
  f:=my_tan(.8)
  print_float()

  WriteF('Next 2 lines should be the same\n')
  f:=.35
  print_float()
  f:=inaccurate(f)
  print_float()
ENDPROC
```

The convert_to_integer and convert_to_float functions perform similar conversions to those done by ! when it occurs in an expression. To make things more explicit, there are a lot of unnecessary uses of !, and these are when f is passed directly as a parameter to a function (in these cases, the ! could safely be omitted). All of the examples have the potential to give different results where they ought to give the same, and this is due to the inaccuracy of floating-point numbers. The last example has been carefully chosen to show this.

1.143 beginner.guide/Accuracy and Range

Accuracy and Range
==================

A floating-point number is just another 32-bit value, so can be stored in LONG variables. It's just the interpretation of the 32-bits which makes them different. A floating-point number can range from numbers as small as 1.3E-38 to numbers as large as 3.4E+38 (that's very small and very large if you don't understand the scientific notation!). However, not every number in this range can accurately be represented, since the number of significant digits is roughly eight.

Accuracy is an important consideration when trying to compare two floating-point numbers and when combining floating-point values after dividing them. It is usually best to check that a floating-point value is in a small range of values, rather than just a particular value. And when combining values, allow for a small amount of error due to rounding etc. See the 'Reference Manual' for more details about the implementation of floating-point numbers.

1.144 beginner.guide/Recursion

Recursion
****

A recursive function is very much like a function which uses a loop. Basically, a recursive function calls itself (usually after some manipulation of data) rather than iterating a bit of code using a loop. There are also recursive types, which are objects with elements which have the object type (in E these would be pointers to objects). We've already seen a recursive type: linked lists, where each element in the list contains a pointer to the next element (see
Linked Lists
).

Recursive definitions are normally much more understandable than an equivalent iterative definition, and it's usually easier to use recursive functions to manipulate this data from a recursive type. However, recursion is by no means a simple topic. Read on at your own peril!

Factorial Example

Mutual Recursion

```
                 Binary Trees

                 Stack (and Crashing)

                 Stack and Exceptions
```

1.145 beginner.guide/Factorial Example

```
                Factorial Example
=================
```

The normal example for a recursive definition is the factorial function, so let's not be different. In school mathematics the symbol ! is used after a number to denote the factorial of that number (and only positive integers have factorials). n! is n-factorial, which is defined as follows:

```
     n! = n * (n-1) * (n-2) * ... * 1     (for n >= 1)
```

So, 4! is 4*3*2*1, which is 24. And, 5! is 5*4*3*2*1, which is 120.

Here's the iterative definition of a factorial function (we'll Raise an exception is the number is not positive, but you can safely leave this check out if you are sure the function will be called only with positive numbers):

```
     PROC fact_iter(n)
       DEF i, result=1
       IF n<=0 THEN Raise("FACT")
       FOR i:=1 TO n
         result:=result*i
       ENDFOR
     ENDPROC result
```

We've used a FOR loop to generate the numbers one to n (the parameter to the fact_iter), and result holds the intermediate and final results. The final result is returned, so check that fact_iter(4) returns 24 and fact_iter(5) returns 120 using a main procedure something like this:

```
     PROC main()
       WriteF('4! is \d\n5! is\d\n', fact_iter(4), fact_iter(5))
     ENDPROC
```

If you're really observant you might have noticed that 5! is 5*4!, and, in general, n! is n*(n-1)!. This is our first glimpse of a recursive definition--we can define the factorial function in terms of itself. The real definition of factorial is (the reason why this is the real definition is because the '...' in the previous definition is not sufficiently precise for a mathematical definition):

```
     1! = 1
     n! = n * (n-1)!    (for n > 1)
```

Notice that there are now two cases to consider. The first case is called the base case and gives an easily calculated value (i.e., no recursion is used). The second case is the recursive case and gives a definition in terms of a number nearer the base case (i.e., (n-1) is nearer 1 than n, for n>1). The normal problem people get into when using recursion is they forget the base case. Without the base case the definition is meaningless. Without a base case in a recursive program the machine is likely to crash! (See

Stack (and Crashing)
.)

We can now define the recursive version of the fact_iter function (again, we'll use a Raise if the number parameter is not positive):

```
PROC fact_rec(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*fact_rec(n-1)
  ELSE
    Raise("FACT")
  ENDIF
ENDPROC
```

Notice how this looks just like the mathematical definition, and is nice and compact. We can even make a one-line function definition (if we omit the check on the parameter being positive):

```
PROC fact_rec2(n) RETURN IF n=1 THEN 1 ELSE n*fact_rec2(n-1)
```

You might be tempted to omit the base case and write something like this:

```
/* Don't do this! */
PROC fact_bad(n) RETURN n*fact_bad(n-1)
```

The problem is the recursion will never end. The function fact_bad will be called with every number from n to zero and then all the negative integers. A value will never be returned, and the machine will crash after a while. The precise reason why it will crash is given later (see

Stack (and Crashing)
).

1.146 beginner.guide/Mutual Recursion

Mutual Recursion
================

In the previous section we saw the function fact_rec which called itself. If you have two functions, fun1 and fun2, and fun1 calls fun2, and fun2 calls fun1, then this pair of functions are mutually recursive. This extends to any amount of functions linked in this way.

This is a rather contrived example of a pair of mutually recursive functions.

```
PROC f(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*g(n-1)
  ELSE
    Raise("F")
  ENDIF
ENDPROC

PROC g(n)
  IF n=1
    RETURN 2*1
  ELSEIF n>=2
    RETURN 2*n*f(n-1)
  ELSE
    Raise("G")
  ENDIF
ENDPROC
```

Both functions are very similar to the fact_rec function, but g returns double the normal values. The overall effect is that every other value in long version of the multiplication is doubled. So, f(n) computes n*(2*(n-1))*(n-2)*(2*(n-3))*...*2 which probably isn't all that interesting.

1.147 beginner.guide/Binary Trees

Binary Trees
============

This is an example of a recursive type and the effect it has on functions which manipulate this type of data. A binary tree is like a linked list, but instead of each element containing only one link to another element there are two links in each element of a binary tree (which point to smaller trees called branches). The first link points to the left branch and the second points to the right branch. Each element of the tree is called a node and there are two kinds of special node: the start point, called the root of the tree (like the head of a list), and the nodes which do not have left or right branches (i.e., NIL pointers for both links), called leaves. Every node of the tree contains some kind of data (just as the linked lists contained an E-string or E-list in each element). The following diagram illustrates a small tree.

```
                 +------+
                 | Root |
                 +--*---+
                   / \
             Left /   \ Right
                 /     \
```

```
    +------*       *------+
    | Node |       | Node |
    +--*---+       +--*---+
      /              / \
Left /         Left /   \ Right
    /              /     \
+--*---+     +----*-+   +-*----+
| Leaf |     | Leaf |   | Leaf |
+------+     +------+   +------+
```

Notice that a node might have only one branch (it doesn't have to have both the left and the right). Also, the leaves on the example were all at the same level, but this doesn't have to be the case. Any of the leaves could easily have been a node which had a lot of nodes branching off it.

So, how can a tree structure like this be written as an E object? Well, the general outline is this:

```
OBJECT tree
  data
  left:PTR TO tree, right:PTR TO tree
ENDOBJECT
```

The left and right elements are pointers to the left and right branches (which will be tree objects, too). The data element is some data for each node. This could equally well be a pointer, an ARRAY or a number of different data elements.

So, what use can be made of such a tree? Well, a common use is for holding a sorted collection of data that needs to be able to have elements added quickly. As an example, the data at each node could be an integer, so a tree of this kind could hold a sorted set of integers. To make the tree sorted, constraints must be placed on the left and right branches of a node. The left branch should contain only nodes with data that is less than the parent node's data, and, similarly, the right branch should contain only nodes with data that is greater. Nodes with the same data could be included in one of the branches, but for our example we'll disallow them. We are now ready to write some functions to manipulate our tree.

The first function is one which starts off a new set of integers (i.e., begins a new tree). This should take an integer as a parameter and return a pointer to the root node of new tree (with the integer as that node's data).

```
PROC new_set(int)
  DEF root:PTR TO tree
  NEW root
  root.data:=int
ENDPROC root
```

The memory for the new tree element must be allocated dynamically, so this is a good example of a use of NEW. Since NEW clears the memory it allocates all elements of the new object will be zero. In particular, the left and right pointers will be NIL, so the root node will also be a leaf. If the NEW fails a "MEM" exception is raised; otherwise the data is set to the supplied value and a pointer to the root node is returned.

To add a new integer to such a set we need to find the appropriate position to insert it and set the left and right branches correctly. This is because if the integer is new to the set it will be added as a new leaf, and so one of the existing nodes will change its left or right branch.

```
PROC add(i, set:PTR TO tree)
  IF set=NIL
    RETURN new_set(i)
  ELSE
    IF i<set.data
      set.left:=add(i, set.left)
    ELSEIF i>set.data
      set.right:=add(i, set.right)
    ENDIF
    RETURN set
  ENDIF
ENDPROC
```

This function returns a pointer to the set to which it added the integer. If this set was initially empty a new set is created; otherwise the original pointer is returned. The appropriate branches are corrected as the search progresses. Only the last assignment to the left or right branch is significant (all others do not change the value of the pointer), since it is this assignment that adds the new leaf. Here’s an iterative version of this function:

```
PROC add_iter(i, set:PTR TO tree)
  DEF node:PTR TO tree
  IF set=NIL
    RETURN new_set(i)
  ELSE
    node:=set
    LOOP
      IF i<node.data
        IF node.left=NIL
          node.left:=new_set(i)
          RETURN set
        ELSE
          node:=node.left
        ENDIF
      ELSEIF i>node.data
        IF node.right=NIL
          node.right:=new_set(i)
          RETURN set
        ELSE
          node:=node.right
        ENDIF
      ELSE
        RETURN set
      ENDIF
    ENDLOOP
  ENDIF
ENDPROC
```

As you can see, it’s quite a bit messier. Recursive functions work well with manipulation of recursive types.

Another really neat example is printing the contents of the set. It’s deceptively simple:

```
PROC show(set:PTR TO tree)
  IF set<>NIL
    show(set.left)
    WriteF('\d ', set.data)
    show(set.right)
  ENDIF
ENDPROC
```

The integers in the nodes will get printed in order (providing they were added using the add function). The left-hand nodes contain the smallest elements so the data they contain is printed first, followed by the data at the current node, and then that in the right-hand nodes. Try writing an iterative version of this function if you fancy a really tough problem.

Putting everything together, here’s a main procedure which can be used to test the above functions:

```
PROC main() HANDLE
  DEF s, i, j
  Rnd(-999999)    /* Initialise seed */
  s:=new_set(10)  /* Initialise set s to contain the number 10 */
  WriteF('Input:\n')
  FOR i:=1 TO 50  /* Generate 50 random numbers and add them to set s */
    j:=Rnd(100)
    add(j, s)
    WriteF('\d ',j)
  ENDFOR
  WriteF('\nOutput:\n')
  show(s)         /* Show the contents of the (sorted) set s */
  WriteF('\n')
EXCEPT
  IF exception="NEW" THEN WriteF('Ran out of memory\n')
ENDPROC
```

1.148 beginner.guide/Stack (and Crashing)

```
                Stack (and Crashing)
====================
```

When you call a procedure you use up a bit of the program’s stack. The stack is used to keep track of procedures in a program which haven’t finished, and real problems can arise when the stack space runs out. Normally, the amount of stack available to each program is sufficient, since the E compiler handles all the fiddly bits quite well. However, programs which use a lot of recursion can quite easily run out of stack.

For example, the fact_rec(10) will need enough stack for ten calls of fact_rec, nine of which are recursively called. This is because each call does not finish until the return value has been computed, so all recursive calls up to fact_rec(1) need to be kept on the stack until fact_rec(1) returns one. Then each procedure will be taken off the stack as they finish. If you try to compute fact_rec(40000), not only will this take a long time, but it will probably run out of stack space. When it does run out of stack, the machine will probably crash or do other weird things. The iterative version, fact_iter does not have these problems, since it only takes one procedure call to calculate a factorial using this function.

If there is the possibility of running out of stack space you can use the FreeStack (built-in) function call (see
System support functions
).
This returns the amount of free stack space. If it drops below about 1KB then you might like to stop the recursion or whatever else is using up the stack. Also, you can specify amount of stack your program gets (and override what the compiler might decide is appropriate) using the OPT STACK option. See the ‘Reference Manual’ for more details on E’s stack organisation.

1.149 beginner.guide/Stack and Exceptions

Stack and Exceptions
====================

The concept ‘recent’ used earlier is connected with the stack (see

Raising an Exception
). A recent procedure is one which is on the stack, the most recent being the current procedure. So, when Raise is called it looks through the stack until it finds a procedure with an exception handler. That handler will then be used, and all procedures before the selected one on the stack are taken off the stack.

Therefore, a recursive function with an exception handler can use Raise in the handler to call the handler in the previous (recursive) call of the function. So anything that has been recursively allocated can be ‘recursively’ deallocated by exception handlers. This is a very powerful and important feature of exception handlers.

1.150 beginner.guide/Object Oriented E

Object Oriented E
****

The Object Oriented Programming (OOP) aspects of E are covered in this chapter. Don’t worry if you don’t know the OOP buzz words like ‘object’, ‘method’ and ‘inheritance’: these terms are explained in the OOP introduction, below. (For some reason, computer science uses strange words to cloak simple concepts in secrecy.)

OOP Introduction

Objects in E

Methods in E

Inheritance in E

Data-Hiding in E

1.151 beginner.guide/OOP Introduction

```
                OOP Introduction
================
```

'Object Oriented Programming' is the name given to a collection of programming techniques that are meant to speed up development and ease maintenance of large programs. These techniques have been around for a long time, but it is only recently that languages that explicitly support them have become popular. You do not need to use a language that supports OOP to program in an Object Oriented way; it's just a bit simpler if you do!

Classes and methods

Example class

Inheritance

1.152 beginner.guide/Classes and methods

```
Classes and methods
-------------------
```

The heart of OOP is the 'Black Box' approach to programming. The kind of black box in question is one where the contents are unknown but there is a number of wires on the outside which give you some way of interacting with the stuff on the inside. The black boxes of OOP are actually collections of data (just like the idea of variables that we've already met) and they are called objects (this is the general term, which is, coincidentally, connected with the OBJECT type in E). Objects can be grouped together in classes, like the types for variables, except that a class also defines what different kinds of wires protrude from the black box. This extra bit (the wires) is known as the interface to the object, and is made up of a number of methods (so a method is analogous to a wire). Each method is actually just like a procedure. With a real black box, the wires are the only way of interacting with the box, so the methods of an object ought to be the only way of creating and using the object. Of course, the methods themselves normally need to know the internal workings of the object, just like the way the wires are normally connected to something inside the black box.

There are two special kinds of methods: constructors and destructors. A constructor is a method which is used to initialise the data in an object, and a class may have several different constructors (allowing for different kinds of initialisation) or it may have none if no special initialisation is necessary. Constructors are normally used to allocate the resources (such as memory) that an object needs. The deallocation of such resources is done by the destructor, of which there is at most one for each class.

Protecting the contents of an object in the 'black box' way is known as data-hiding (the data in the object is visible only to its methods), and only allowing the contents of an object to be manipulated via its interface is known as data abstraction. By using this approach, only the methods know the structure of the data in an object and so this structure can be changed without affecting the whole of a program: only the methods would potentially need recoding. As you might be able to tell, this simplifies maintenance quite considerably.

1.153 beginner.guide/Example class

Example class

A good example of a class is the mathematical notion of a set (of integers). A particular object from this class would represent a particular set of integers. The interface for the class would probably include the following methods:

1. Add -- adds an integer to a set object.

2. Member -- tests for membership of an integer in a set object.

3. Empty -- tests for emptiness of a set object.

4. Union -- unions a set object with a set object.

A more complete class would also contain methods for removing elements, intersecting sets etc. The important thing to notice is that to use this class you need to know only how to use the methods. The black box approach means that we don't (and shouldn't) know how the set class is actually implemented, i.e., how data is structured within a set object. Only the methods themselves need to know how to manipulate the data that represents a set object.

The benefit of OOP comes when you actually use the classes, so suppose you implement this set class and then use it in your code for some database program. If you found that the set implementation was a bit inefficient (in terms of memory or speed), then, since you programmed in this OOP way, you wouldn't have to recode the whole database program, just the set class! You can change the way the set data is structured in an object as much and as often as you like, so long as each implementation has the same interface (and gives the same results!).

1.154 beginner.guide/Inheritance

Inheritance

The remaining OOP concept of interest is inheritance. This is a grand name for a way of building on classes that enables the derived (i.e., bigger) class to be used as if its objects were really members of the inherited, or base, class. For example, suppose class D were derived from class B, so D is the derived class and B is the base class. In this case, class D inherits the data structure of class B, and may add extra data to it. It also inherits all the methods of class B, and objects of class D may be treated as if they were really objects of class B.

Of course, an inherited method cannot affect the extra data in class D, only the inherited data. To affect the extra data, class D can have extra methods defined, or it can make new definitions for the inherited methods. The latter approach is only really useful if the new definition of an inherited method is pretty similar to the inherited method, differing only in how it affects the extra data in class D. This overriding of methods does not affect the methods in class B (nor those of other classes derived from B), but only those in class D and the classes derived from D.

1.155 beginner.guide/Objects in E

Objects in E
============

Classes are defined using OBJECT in the same way that we've seen before (see
OBJECT Type
). So, in E, the terms 'object declaration' and 'class' may be used interchangeably. However, referring to an OBJECT type as a 'class' signals the presence of methods in an object.

The following example OBJECT is the basis of a set class, as described above (see
Example class
). This set implementation is going to be quite simple and it will be limited to a maximum of 100 elements.

```
     OBJECT set
```

```
    elts[100]:ARRAY OF LONG
    size
  ENDOBJECT
```

Currently, the only way to allocate an OOP object is to use NEW with an appropriately typed pointer. The following sections of code all allocate memory for the data of set, but only the last one allocates an OOP set object. Each one may use and access the set data, but only the last one may call the methods of set.

```
DEF s:set

DEF s:PTR TO set
s:=NewR(SIZEOF set)

DEF s:PTR TO set
s:=NEW s
```

OOP objects can, of course, be deallocated using END, in which case the destructor for the corresponding class is also called. Leaving an OOP object to be deallocated automatically at the end of the program is not quite as safe as normal, since in this case the destructor will not be called. Constructors and destructors are described in more detail below.

1.156 beginner.guide/Methods in E

Methods in E
============

The methods of E are very similar to normal procedures, but there is one, big difference: a method is part of a class, so must somehow be identified with the other parts of the class. In E this identification is done by relating all methods to the corresponding OBJECT type for the class, using the OF keyword after the description of the method's parameters. So, the methods of the simple set class would be defined as outlined below (of course, these examples have omitted the code of methods).

```
PROC add(x) OF set
  /* code for add method */
ENDPROC

PROC member(x) OF set
  /* code for member method */
ENDPROC

PROC empty() OF set
  /* code for empty method */
ENDPROC

PROC union(s:PTR TO set) OF set
  /* code for union method */
ENDPROC
```

At first sight it might seem that the particular set object which would be manipulated by these methods is missing from the parameters. For instance, it appears that the empty method should need an extra PTR TO set parameter, and that would be the set object it tested for emptiness. However, methods are called in a slightly different way to normal procedures. A method is a part of a class, and is called in a similar way to accessing the data elements of the class. That is, the method is selected using . and acts (implicitly) on the object from which it was selected. The following example shows the allocation of a set object and the use of some of the above methods.

```
DEF s:PTR TO set
NEW s  -> Allocate an OOP object
s.add(17)
s.add(-34)
IF s.empty()
  WriteF('Error: the set s should not be empty!\n')
ELSE
  WriteF('OK: not empty\n')
ENDIF
IF s.member(0)
  WriteF('Error: how did 0 get in there?\n')
ELSE
  WriteF('OK: 0 is not a member\n')
ENDIF
IF s.member(-34)
  WriteF('OK: -34 is a member\n')
ELSE
  WriteF('Error: where has -34 gone?\n')
ENDIF
END s  -> Finished with s now
```

This is why the methods do not take that extra PTR TO set argument. If a method is called then it has been selected from an appropriate object, and so this must be the object which it affects. The slightly complicated method is union which adds another set object by unioning it. In this case, the argument to the method is a PTR TO set, but this is the set to be added, not the set which is being expanded.

So, how do you refer to the object which is being affected? In other words, how do you affect it? Well, this is the remaining difference from normal procedures: every method has a special local variable, self, which is of type PTR TO class and is initialised to point to the object from which the method was selected. Using this variable, the data and methods of object can be accessed and used as normal. For instance, the empty method has a self local variable of type PTR TO set, and can be defined as below:

```
PROC empty() OF set IS self.size=0
```

Constructors are simply methods which initialise the data of an object. For this reason they should normally be called only when the object is allocated. The NEW operator allows OOP objects to call a constructor at the point at which they are allocated, to make this easier and more explicit. The constructor will be called after NEW has allocated the memory for the object. It is wise to give constructors suggestive names like create and copy, or the same name as the class. The following constructors might be defined for the set class:

```
/* Create empty set */
PROC create() OF set
  self.size=0
ENDPROC

/* Copy existing set */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  FOR i:=0 TO oldset.size-1
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC
```

They would be used as in the code below. Notice that the create constructor is, in this case, redundant since NEW will initialise the data elements to zero. If NEW does sufficient initialisation then you do not have to define any constructors, and even if you do have constructors you don’t have to use them when allocating objects.

```
DEF s:PTR TO set, t:PTR TO set, u:PTR TO set
NEW s.create()
IF s.empty THEN WriteF('s is empty\n')
END s
NEW t  /* This happens to be the same as using create */
IF t.empty THEN WriteF('t is empty\n')
t.add(10)
NEW u.copy(t)
IF u.member(10) THEN WriteF('10 is in u\n')
END t, u
```

For each class there is at most one destructor, and this is responsible for clearing up and deallocating resources. If one is needed then it must be called end, and (as this might suggest) it is called automatically when an OOP object is deallocated using END. So, for OOP objects with a destructor, the (roughly) equivalent code to END using Dispose is a bit different. Take care to note that the destructor is not called if END is not used to deallocate an OOP object (i.e., if deallocation is left to be done automatically at the end of the program).

```
END p

IF p
  p.end()  -> Call destructor
  Dispose(p)
  p:=NIL
ENDIF
```

The simple implementation of the set class needs no destructor. If, however, the elements data were a pointer (to LONG), and the array were allocated based on some size parameter to a constructor, then a destructor would be useful. In this case the set class would also need a maxsize data element, which records the maximum, allocated size of the elements array.

```
OBJECT set
  elements:PTR TO LONG
  size
  maxsize
ENDOBJECT

PROC create(sz=100) OF set  -> Default to 100
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100
  self.elements:=NEW p[self.maxsize]
ENDPROC

PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize=0
    WriteF('Error: did not create() the set\n')
  ELSE
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC
```

Without the destructor end, the memory allocated for elements would not be deallocated when END is used, although it would get deallocated at the end of the program (in this case). However, if AllocMem were used instead of NEW to allocate the array, then the memory would have to be deallocated using FreeMem, and this would best be done in the destructor, as above. (The memory would not be deallocated automatically at the end of the program if AllocMem is used.) Another solution to this kind of problem would be to have a special method which called FreeMem, and to remember to call this method just before deallocating one of these objects, so you can see that the interaction of END with destructors is quite useful.

Already, the above re-definition of set begins to show the power of OOP. The actual implementation of the set class is very different, but the interface can remain the same. The code for the methods would need to change to take into account the new maxsize element (where before the fixed size of 100 was used), and also to deal with the possibility the create constructor had not been used (in which case elements would be NIL and maxsize zero). But the code which used the set class would not need to change, except maybe to allocate more sensibly sized sets!

Yet another, different implementation of a set was outlined above (see

Binary Trees

). In fact, remarkably few changes would be needed to convert the code from that section into another implementation of the set class. The new_set procedure is like a set constructor which initialises the set to be a singleton (i.e., to contain one element), and the add procedure is just like the add method of the set class. The only slight problem is that empty sets are not modelled by the binary tree implementation, so it wouldn't, as it stands, be a complete implementation. It would be straight-forward (but unduly complicated at this point) to add support for empty sets to this particular implementation.

1.157 beginner.guide/Inheritance in E

```
                Inheritance in E
================
```

One class is derived from another using the OF keyword in the definition of the derived class OBJECT, in a similar way that OF is used with methods. For instance, the following code shows how to define the class d to be derived from class b. The class b is then said to be inherited by the class d.

```
     OBJECT b
       b_data
     ENDOBJECT

     OBJECT d OF b
       extra_d_data
     ENDOBJECT
```

The names b and d have been chosen to be somewhat suggestive, since the class which is inherited (i.e., b) is known as the base class, whilst the inheriting class (i.e., d) is known as the derived class.

The definition of d is the same as the following definition of duff, except for one major difference: with the above derivation the methods of b are also inherited by d and they become methods of class d. The definition of duff relates it in no way to b, except at best accidentally (since any changes to b do not affect duff, whereas they would affect d).

```
     OBJECT duff
       b_data
       extra_d_data
     ENDOBJECT
```

One property of this derivation applies to the data records built by OBJECT as well as the OOP classes. The data records of type d or duff may be used wherever a data record of type b were required (e.g., the argument to some procedure), and they are, in fact, indistinguishable from records of type b. Although, if the definition of b were changed (e.g., by changing the name of the b_data element) then data records of type duff would not be usable in this way, but those of type d still would. Therefore, it is wise to use inheritance to show the relationships between classes or data of OBJECT types. The following example shows how procedure print_b_data can validly be called in several ways, given the definitions of b, d and duff above.

```
     PROC print_b_data(p:PTR TO b)
       WriteF('b_data = \d\n', p.b_data)
     ENDPROC

     PROC main()
       DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
       NEW p_b, p_d, p_duff
```

```
  p_b.b_data:=11
  p_d.b_data:=-3
  p_duff.b_data:=27
  WriteF('Printing p_b: ')
  print_b_data(p_b)
  WriteF('Printing p_d: ')
  print_b_data(p_d)
  WriteF('Printing p_duff: ')
  print_b_data(p_duff)
ENDPROC
```

So far, no methods have been defined for b, which means that it is just an OBJECT type. The procedure print_b_data suggests a useful method of b, which will be called print.

```
PROC print() OF b
  WriteF('b_data = \d\n', self.b_data)
ENDPROC
```

This definition would also define a print method for d, since d is derived from b and it inherits all the methods of b. However, duff would, of course, still be just an OBJECT type, although it could have a similar print method explicitly defined for it. If b has any methods defined for it (i.e., if it is a class) then data records of type duff cannot be used as if they were objects of the class b, and it is not safe to try! In this case, only objects of derived class d can be used in this manner. (If b is a class then d is a class, due to inheritance.)

```
PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
  NEW p_b, p_d, p_duff
  p_b.b_data:=11
  p_d.b_data:=-3;   p_d.extra_d_data:=3
  p_duff.b_data:=7; p_duff.extra_d_data:=-7
  WriteF('Printing p_b: ')
/* b explicitly has print method  */
  p_b.print()
  WriteF('Printing p_d: ')
/* d inherits print method from b */
  p_d.print()
  WriteF('No print method for p_duff\n')
/* Do not try to print p_duff in this way */
/*  p_duff.print() */
ENDPROC
```

Unfortunately, the print method inherited by d only prints the b_data element (since it is really a method of b, so cannot access the extra data added in d). However, any inherited method can be overridden by defining it again, this time for the derived class.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  WriteF('b_data = \d\n', self.b_data)
ENDPROC
```

With this extra definition, the same main procedure above would now print all the data of d, but only the b_data element of b. This is because the new definition of print affects only class d (and classes derived from d).

Inherited methods are often overridden just to add extra functionality, as in the case above where we wanted the extra data to be printed as well as the data derived from b. For this purpose, the SUPER operator can be used on a method call to force the base class method to be used, where normally the derived class method would be used. So, the definition of the print method for class d could call the print method of class b.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  SUPER self.print()
ENDPROC
```

Be careful, though, because without the SUPER operator this would involve a recursive call to the print method of class d, rather than a call to the base class method.

Just as data records of type d can be used wherever data records of type b were required, objects of class d can used in place of objects of class b. The following procedure prints a message and the object data, using the print method of b. (Of course, only the methods named by class b can be used in such a procedure, since the pointer p is of type PTR TO b.)

```
PROC msg_print(msg, p:PTR TO b)
  WriteF('Printing \s: ', msg)
  p.print()
ENDPROC

PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d
  NEW p_b, p_d
  p_b.b_data:=11
  p_d.b_data:=-3; p_d.extra_d_data:=3
  msg_print('p_b', p_b)
  msg_print('p_d', p_d)
ENDPROC
```

You can't use duff now, since it is not a class and b is, and msg_print expects a pointer to class b. The only other objects that can be passed to msg_print are objects from classes derived from b, and this is why p_d can be printed using msg_print. If you collect together the code and run the example you will see that the call to print in msg_print uses the overridden print method when msg_print is called with p_d as a parameter. That is, the correct method is called even though the pointer p is not of type PTR TO d. This is called polymorphism: different implementations of print may be called depending on the real, dynamic type of p. Here's what should be printed:

```
Printing p_b: b_data = 11
Printing p_d: extra_d_data = 3, b_data = -3
```

Inheritance is not limited to a single layer: you can derive other classes from b, you can derive classes from d, and so on. For instance, if class e is derived from class d then it would inherit all the data of d and all the methods of d. This means that e would inherit the richer version of print, and may even override it yet again. In this case, class e would have two base classes, b and d, but would be derived directly from d (and indirectly from b, via d). Class d would therefore be known as the super class of e, since e is derived directly from d. (The super class of d is its only base class, b.) So, the SUPER operator is actually used to call the methods in the super class. In this example, the SUPER operator can be used in the methods of e to call methods of d.

The binary tree implementation above (see Binary Trees) suggests a good example for a class hierarchy (a collection of classes related by inheritance). A basic tree structure can be encapsulated in a base class definition, and then specific kinds of tree (with different data at the nodes) can be derived from this. In fact, the base class tree defined below is only useful for inheriting, since a tree is pretty useless without some data attached to the nodes. Since it is very likely that objects of class tree will never be useful (but objects of classes derived from tree would be), the tree class is called an abstract class.

```
OBJECT tree
  left:PTR TO tree, right:PTR TO tree
ENDOBJECT

PROC nodes() OF tree
  DEF tot=1
  IF self.left  THEN tot:=tot+self.left.nodes()
  IF self.right THEN tot:=tot+self.right.nodes()
ENDPROC tot

PROC leaves(show=FALSE) OF tree
  DEF tot=0
  IF self.left
    tot:=tot+self.left.leaves(show)
  ENDIF
  IF self.right
    tot:=tot+self.right.leaves(show)
  ELSEIF self.left=NIL
    IF show THEN self.print_node()
    tot++
  ENDIF
ENDPROC tot

PROC print_node() OF tree
  WriteF('<NULL> ')
ENDPROC

PROC print() OF tree
  IF self.left  THEN self.left.print()
  self.print_node()
  IF self.right THEN self.right.print()
ENDPROC
```

The nodes and leaves methods return the number of nodes and leaves of the tree, respectively, with the leaves method taking a flag to specify whether the leaves should also be printed. These methods should never need overriding in a class derived from tree, and neither should print, which traverses the tree, printing the nodes from left to right. However, the print_node method probably should be overridden, as is the case in the integer tree defined below.

```
OBJECT integer_tree OF tree
  int
ENDOBJECT

PROC create(i) OF integer_tree
  self.int:=i
ENDPROC

PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      p:=self.left
      p.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      p:=self.right
      p.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC

PROC print_node() OF integer_tree
  WriteF('\d ', self.int)
ENDPROC
```

This is a nice example of polymorphism at work: we can implement a tree which works with integers simply by defining the appropriate methods. The leaves method (of the tree class) will then automatically call the integer_tree version of print_node whenever we pass it an integer_tree object. The definitions of tree and integer_tree can even be in different modules (see
Data-Hiding in E
), and, using these OOP techniques, the module containing tree would not need to be recompiled even if a class like integer_tree is added or changed. This shows why OOP is good for code-reuse and extensibility: with traditional programming techniques we would have to adapt the binary tree functions to account for integers, and again for each new datatype.

Notice that the recursive use of the new method add must be called via an auxiliary pointer, p, of the derived class. This is because the left and right elements of tree are pointers to tree objects and add is not a method of tree (the compiler would reject the code as a syntax error if you tried to directly access add under these circumstances). Of course, if the tree class had an add method there would not be this problem, but what would the code be for such a method?

An add method does not really make sense for tree, but if almost all classes derived from tree are going to need such a method it might be nice to include it in the tree base class. This is the purpose of abstract methods. An abstract method is one which exists in a base class solely so that it can be overridden in some derived class. Normally, such methods have no sensible definition in the base class, so there is a special keyword, EMPTY, which can be used to define them. For example, the add method in tree would be defined as below.

```
PROC add(x) OF tree IS EMPTY
```

With this definition, the code for the add method for the integer_tree class could be simplified. (The auxiliary pointer, p, is still needed for use with NEW, since an expression like self.left is not a pointer variable.)

```
PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      self.left.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      self.right.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC
```

This, however, is not the best example of an abstract method, since the add method in every class derived from tree must now take a single LONG value as an parameter, in order to be compatible. In general, though, a class representing a tree with node data of type t would really want an add method to take a single parameter of type t. The fact that a LONG value can represent a pointer to any type is helpful, here. This means that the definition of add may not be so limiting, after all.

The print_node method is much more obviously suited to being an abstract method. The above definition prints something silly, because at that point we didn't know about abstract methods and we needed the method to be defined in the base class. A much better definition would make print_node abstract.

```
PROC print_node() OF tree IS EMPTY
```

It is quite safe to call these abstract methods, even for tree class objects. If a method is still abstract in any class (i.e., it has not been overridden), then calling it on objects of that class has the same effect as calling a function which just returns zero (i.e., it does very little!).

The integer_tree class could be used like this:

```
PROC main()
  DEF t:PTR TO integer_tree
  NEW t.create(10)
  t.add(-10)
  t.add(3)
  t.add(5)
  t.add(-1)
  t.add(1)
  WriteF('t has \d nodes, with \d leaves: ',
         t.nodes(), t.leaves())
  t.leaves(TRUE)
  WriteF('\n')
  WriteF('Contents of t: ')
  t.print()
  WriteF('\n')
  END t
ENDPROC
```

1.158 beginner.guide/Data-Hiding in E

```
Data-Hiding in E
================
```

Data-hiding is accomplished in E at the module level. This means, effectively, that it is wise to define classes in separate modules (or at least only closely related classes together in a module), taking care to EXPORT only the definitions that you need to. You can also use the PRIVATE keyword in the definition of any OBJECT to hide all the elements following it from code which uses the module (although this does not affect the code within the module). The PUBLIC keyword can be used in a similar way to make the elements which follow visible (i.e., accessible) again, as they are by default. For instance, the following OBJECT definition makes x, y, a and b private (so only visible to the code within the same module), and p, q and r public (so visible to code external to the module, too).

```
OBJECT rec
  p:INT
PRIVATE
  x:INT
  y
PUBLIC
  q
  r:PTR TO LONG
PRIVATE
  a:PTR TO LONG, b
ENDOBJECT
```

For the set class you would probably want to make all the data private and all the methods public. In this way you force programs which use this module to use the supplied interface, rather than fiddling with the set data structures themselves. The following example is the complete code for a simple, inefficient set class, and can be compiled to a module.

```
OPT MODULE  -> Define class 'set' in a module
OPT EXPORT  -> Export everything

/* The data for the class */
OBJECT set PRIVATE  -> Make all the data private
  elements:PTR TO LONG
  maxsize, size
ENDOBJECT

/* Creation constructor */
/* Minimum size of 1, maximum 100000, default 100 */
PROC create(sz=100) OF set
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100 -> Check size
  self.elements:=NEW p[self.maxsize]
ENDPROC

/* Copy constructor */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  self.create(oldset.maxsize)  -> Call create method!
  FOR i:=0 TO oldset.size-1  -> Copy elements
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC

/* Destructor */
PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize<>0  -> Check that it was allocated
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC

/* Add an element */
PROC add(x) OF set
  IF self.member(x)=FALSE  -> Is it new? (Call member method!)
    IF self.size=self.maxsize
      Raise("full")  -> The set is already full
    ELSE
      self.elements[self.size]:=x
      self.size:=self.size+1
    ENDIF
  ENDIF
ENDPROC

/* Test for membership */
PROC member(x) OF set
  DEF i
  FOR i:=0 TO self.size-1
    IF self.elements[i]=x THEN RETURN TRUE
  ENDFOR
ENDPROC FALSE
```

```
    /* Test for emptiness */
    PROC empty() OF set IS self.size=0

    /* Union (add) another set */
    PROC union(other:PTR TO set) OF set
      DEF i
      FOR i:=0 TO other.size-1
        self.add(other.elements[i])  -> Call add method!
      ENDFOR
    ENDPROC

    /* Print out the contents */
    PROC print() OF set
      DEF i
      WriteF('{ ')
      FOR i:=0 TO self.size-1
        WriteF('\d ', self.elements[i])
      ENDFOR
      WriteF('}')
    ENDPROC
```

   This class can be used in another module or program, as below:

```
    MODULE '*set'

    PROC main() HANDLE
      DEF s=NIL:PTR TO set
      NEW s.create(20)
      s.add(1)
      s.add(-13)
      s.add(91)
      s.add(42)
      s.add(-76)
      IF s.member(1) THEN WriteF('1 is a member\n')
      IF s.member(11) THEN WriteF('11 is a member\n')
      WriteF('s = ')
      s.print()
      WriteF('\n')
    EXCEPT DO
      END s
      SELECT exception
      CASE "NEW"
        WriteF('Out of memory\n')
      CASE "full"
        WriteF('Set is full\n')
      ENDSELECT
    ENDPROC
```

1.159 beginner.guide/Introduction to the Examples

Introduction to the Examples
****

In this part we shall go through some slightly larger examples than those in the previous parts. However, none of them are too big, so they should still be easy to understand. The note-worthy parts of each example are described, and you may even find the odd comment in the code. Large, complicated programs benefit hugely from the odd well-placed and descriptive comment. This fact can't be stressed enough.

All the examples will run on a standard Amiga, except for the one which uses ReadArgs (an AmigaDOS 2.0 function). It is really worth upgrading your system to AmigaDOS 2.0 (or above) if you are still using previous versions. The ReadArgs example can only hint at the power and friendliness of the newer system functions. If you are fortunate enough to have an A4000 or an accelerated machine, then the timing example will give better (i.e., quicker) results.

Supplied with this Guide should be a directory of sources of most of the examples. Here's a complete catalogue:

simple.e
The simple program from the introduction. See
A Simple Program
.

while.e
The slightly complicated WHILE loop. See
WHILE loop
.

address.e
The program which prints the addresses of some variables. See

Finding addresses (making pointers)
.

static.e
The static data problem. See
Static data
.

static2.e
The first solution to the static data problem. See
Static data
.

except.e
An exception handler example. See
Raising an Exception
.

except2.e
Another exception handler example. See
Raising an Exception
.

static3.e
The second solution to the static data problem, using NEW. See

List and typed list allocation

.

float.e
The floating-point example program. See
Floating-Point Functions
.

bintree.e
The binary tree example. See
Binary Trees
.

tree.e
The tree and integer_tree classes, as a module. See
Inheritance in E
.

tree-use.e
A program to use the integer_tree class. See
Inheritance in E
.

set.e
The simple, inefficient set class, as a module. See
Data-Hiding in E
.

set-use.e
A program to use the set class. See
Data-Hiding in E
.

csv-estr.e
The CSV reading program using E-strings. See
String Handling and I-O
.

csv-norm.e
The CSV reading program using normal strings. See

String Handling and I-O
.

csv-buff.e
The CSV reading program using normal strings and a large buffer. See

String Handling and I-O
.

csv.e
The CSV reading program using normal strings, a large buffer, and an exception handler. See
String Handling and I-O
.

timing.e
The timing example. See

Timing Expressions
.

args.e
The argument parsing example for any AmigaDOS. See
Any AmigaDOS
.

args20.e
The argument parsing example for any AmigaDOS 2.0 and above. See

AmigaDOS 2.0 (and above)
.

gadgets.e
The gadgets example. See
Gadgets
.

idcmp.e
The IDCMP and gadgets example. See
IDCMP Messages
.

graphics.e
The graphics example. See
Graphics
.

screens.e
The screens example, without an exception handler. See
Screens
.

screens2.e
The screens example again, but this time with an exception handler.
See
Screens
.

dragon.e
The dragon curve recursion example. See
Recursion Example
.

1.160 beginner.guide/String Handling and I-O

String Handling and I/O
****

This chapter shows how to use normal strings and E-strings, and also how to read data from a file. The programs use a number of the string functions and make effective (but different) use of memory where possible.

The key points to understand are:

- The difference between normal strings and E-strings.

- The two methods of reading data from a file (line-by-line or all at once).

- The necessary allocation of memory for E-strings.

- The unnecessary, but advisable, deallocation of the E-string memory once it is no longer needed. The deallocation could be left to the automatic deallocation at the end of the program, but that would waste an increasing amount of memory whilst the program was running. If the input data was large then memory could easily be exhausted.

- The way in which sections of an E-string (or a normal string, for that matter) can easily be turned into normal strings.

- The way exception handlers can tidy up programs.

The problem to solve is reading of a CSV (comma separated variables) file, which is a standard format file for databases and spreadsheets. The format is very simple: each record is a line (i.e., terminated with a line-feed) and each field in a record is separated by a comma. To make this example a lot simpler, we will forbid a field to contain a comma (normally this would require the field to be quoted). So, a typical input file would look like this:

```
Field1,Field2,Field3
10,19,-3
fred,barney,wilma
,,last
first,,
```

In this example all records have three fields, as is well illustrated by the first line (i.e., the first record). The last two records may seem a bit strange, but they just show how fields can be blank. In the last record all but the first field are blank, and in the previous record all but the last are blank.

So now we know the format of the file to be read. To operate on a file we must first open it using the Open function (from the dos.library), and to read the lines from the file we will use the ReadStr (built-in) function. There will be four versions of a program to read a CSV file: two of which read data line-by-line and two which read all the file at once. Of the two which read line-by-line, one manipulates the read lines as E-strings and the other uses normal strings. The use of normal strings is arguably more advanced than the use of E-strings, since cunning tricks are employed to make effective use of memory. However, the programs are not meant to show that E-strings are better than normal strings (or vice versa), rather they are meant to show how to use strings properly.

```
/* A suitably large size for the record buffer */
CONST BUFFERSIZE=512

PROC main()
  DEF filehandle, status, buffer[BUFFERSIZE]:STRING, filename
```

```
    filename:='datafile'
    IF filehandle:=Open(filename, OLDFILE)
      REPEAT
        status:=ReadStr(filehandle, buffer)
        /* This is the way to check ReadStr() actually read something */
        IF buffer[] OR (status<>-1) THEN process_record(buffer)
      UNTIL status=-1
      /* If Open() succeeded then we must Close() the file */
      Close(filehandle)
    ELSE
      WriteF('Error: Failed to open "\s"\n', filename)
    ENDIF
  ENDPROC

  PROC process_record(line)
    DEF i=1, start=0, end, len, s
    /* Show the whole line being processed */
    WriteF('Processing record: "\s"\n', line)
    REPEAT
      /* Find the index of a comma after the start index */
      end:=InStr(line, ',', start)
      /* Length is end index minus start index */
      len:=(IF end<>-1 THEN end ELSE EstrLen(line))-start
      IF len>0
        /* Allocate an E-string of the correct length */
        IF s:=String(len)
          /* Copy the portion of the line to the E-string s */
          MidStr(s, line, start, len)
          /* At this point we could do something useful... */
          WriteF('\t\d) "\s"\n', i, s)
          /* We've finished with the E-string so deallocate it */
          DisposeLink(s)
        ELSE
          /* It's a non-fatal error if the String() call fails */
          WriteF('\t\d) Memory exhausted! (len=\d)\n', len)
        ENDIF
      ELSE
        WriteF('\t\d) Empty Field\n', i)
      ENDIF
      /* The new start is after the end we found */
      start:=end+1
      INC i
    /* Once a comma is not found we've finished */
    UNTIL end=-1
  ENDPROC
```

There are a couple of points worth noting about this program:

* A large E-string, buffer, is used to hold each line before it is processed. If a record exceeds the size of this E-string then ReadStr will only read a partial record, and the next ReadStr will read some more this record. However, the program considers each call to ReadStr to read a whole record, so it will get the records slightly wrong in this case. This is a limitation of the program and it should be documented so that users know to constrain themselves to datafiles without long lines.

- The file name is 'hard-wired' to be datafile. A more flexible program would allow this to be passed as an argument (see

  Argument Parsing
).

- ReadStr may return -1 to indicate an error (usually when the end of the file has been reached), but the E-string read so far may still be valid. The check on the E-string and error value is the proper way of deciding whether ReadStr actually read anything from the file.

- Look carefully at the manipulation of the string indexes start and end, and the calculation of the length of a portion of a string.

- MidStr is used to copy a field from a record, so an E-string must be used to hold the field.

- The E-string s is only valid between the successful allocation by string and the DisposeLink. It would be incorrect to try to, for instance, print it at any other point. On the other hand, a more complicated program may want to store up all the data, and so it may be inappropriate to deallocate the E-string at this point. In this case, the pointer to the E-string could be stored and it might be valid for the rest of the program.

- The allocation using String is very closely followed by deallocation using DisposeLink. This suggests that a single E-string could be allocated and used repeatedly (like buffer is), due to the simple nature of this example.

To change this to use normal strings (in a very memory efficient way), we need to alter only the process_record procedure. Some note-worthy differences are:

- Small parts of the E-string buffer are turned into normal strings by terminating them with NIL when necessary. This involves changing a comma that is found.

- No new memory is allocated, rather the buffer memory is reused (as described above). This is fine for this example, although if the fields were needed after a record had been processed they would need to be copied, since the contents of buffer are changed by ReadStr.

```
PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Show the whole line being processed */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Find the index of a comma after the start index */
    end:=InStr(line, ',', start)
    /* If a comma was found then terminate with a NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Point to the start of the field */
    s:=line+start
    IF s[]
      /* At this point we could do something useful... */
      WriteF('\t\d) "\s"\n', i, s)
```

```
    ELSE
      WriteF('\t\d) Empty Field\n', i)
    ENDIF
    /* The new start is after the end we found */
    start:=end+1
    INC i
  /* Once a comma is not found we've finished */
  UNTIL end=-1
ENDPROC
```

The next two versions of the program are basically the same: they both read the whole file into one large, dynamically allocated buffer and then process the data. The second of the two versions also uses exceptions to make the program much more readable. The differences from the above version which uses normal strings are:

- The main procedure calculates the length of the data in the file and then uses New to dynamically allocate some memory to hold it.

- The read data is terminated with a NIL so that it can safely be treated as a (very long) normal string.

- The process_buffer procedure splits the read data up into lots of normal strings, one for each line of data.

```
PROC main()
  DEF buffer, filehandle, len, filename
  filename:='datafile'
  /* Get the length of data in the file */
  IF 0<(len:=FileLength(filename))
    /* Allocate just enough room for the data + a terminating NIL */
    IF buffer:=New(len+1)
      IF filehandle:=Open(filename, OLDFILE)
        /* Read whole file, checking amount read */
        IF len=Read(filehandle, buffer, len)
          /* Terminate buffer with a NIL just in case... */
          buffer[len]:=NIL
          process_buffer(buffer, len)
        ELSE
          WriteF('Error: File reading error\n')
        ENDIF
        /* If Open() succeeded then we must Close() the file */
        Close(filehandle)
      ELSE
        WriteF('Error: Failed to open "\s"\n', filename)
      ENDIF
      /* Deallocate buffer (not really necessary in this example) */
      Dispose(buffer)
    ELSE
      WriteF('Error: Insufficient memory to load file\n')
    ENDIF
  ELSE
    WriteF('Error: "\s" is an empty file\n', filename)
  ENDIF
ENDPROC

/* buffer is like a normal string since it's NIL-terminated */
```

```
PROC process_buffer(buffer, len)
  DEF start=0, end
  REPEAT
    /* Find the index of a linefeed after the start index */
    end:=InStr(buffer, '\n', start)
    /* If a linefeed was found then terminate with a NIL */
    IF end<>-1 THEN buffer[end]:=NIL
    process_record(buffer+start)
    start:=end+1
  /* We've finished if at the end or no more linefeeds */
  UNTIL (start>=len) OR (end=-1)
ENDPROC

PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Show the whole line being processed */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Find the index of a comma after the start index */
    end:=InStr(line, ',', start)
    /* If a comma was found then terminate with a NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Point to the start of the field */
    s:=line+start
    IF s[]
      /* At this point we could do something useful... */
      WriteF('\t\d) "\s"\n', i, s)
    ELSE
      WriteF('\t\d) Empty Field\n', i)
    ENDIF
    /* The new start is after the end we found */
    start:=end+1
    INC i
  /* Once a comma is not found we've finished */
  UNTIL end=-1
ENDPROC
```

The program is now quite messy, with many error cases in the main procedure. We can very simply change this by using an exception handler and a few automatic exceptions.

```
/* Some constants for exceptions (ERR_NONE is zero: no error) */
ENUM ERR_NONE, ERR_LEN, ERR_NEW, ERR_OPEN, ERR_READ

/* Make some exceptions automatic */
RAISE ERR_LEN  IF FileLength()<=0,
      ERR_NEW  IF New()=NIL,
      ERR_OPEN IF Open()=NIL

PROC main() HANDLE
  /* Note the careful initialisation of buffer and filehandle */
  DEF buffer=NIL, filehandle=NIL, len, filename
  filename:='datafile'
  /* Get the length of data in the file */
  len:=FileLength(filename)
  /* Allocate just enough room for the data + a terminating NIL */
  buffer:=New(len+1)
```

```
    filehandle:=Open(filename, OLDFILE)
    /* Read whole file, checking amount read */
    IF len<>Read(filehandle, buffer, len) THEN Raise(ERR_READ)
    /* Terminate buffer with a NIL just in case... */
    buffer[len]:=NIL
    process_buffer(buffer, len)
  EXCEPT DO
    /* Both of these are safe thanks to the initialisations */
    IF buffer THEN Dispose(buffer)
    IF filehandle THEN Close(filehandle)
    /* Report error (if there was one) */
    SELECT exception
    CASE ERR_LEN;   WriteF('Error: "\s" is an empty file\n', filename)
    CASE ERR_NEW;   WriteF('Error: Insufficient memory to load file\n')
    CASE ERR_OPEN;  WriteF('Error: Failed to open "\s"\n', filename)
    CASE ERR_READ;  WriteF('Error: File reading error\n')
    ENDSELECT
  ENDPROC
```

The code is now much clearer, and the majority of errors can be caught automatically. Notice that the exception handler is called even if the program succeeds (thanks to the DO after the EXCEPT). This is because when the program terminates it needs to deallocate the resources it allocated in every case (successful or otherwise), so the code is the same. Conditional deallocation (of the buffer, for example) is made safe by an appropriate initialisation.

If you feel like a small exercise, try to write a similar program but this time using the tools/file module which comes in the standard Amiga E distribution. Of course, you'll first need to read the accompanying documentation, but you should find that this module makes file interaction very simple.

1.161 beginner.guide/Timing Expressions

Timing Expressions
****

You may recall the outline of a timing procedure in Part Two (see

Evaluation
). This chapter gives the complete version of this example. The information missing from the outline was how to determine the system time and use this to calculate the time taken by calls to Eval. So the things to notice about this example are:

- Use of the Amiga system function DateStamp (from the dos.library). (You really need the 'Rom Kernel Reference Manuals' and the 'AmigaDOS Manual' to understand the system functions.)

- Use of the module dos/dos to include the definitions of the object datestamp and the constant TICKS_PER_SECOND. (There are fifty ticks per second.)

- Use of the repeat procedure to do Eval a decent number of times for each expression (so that some time is taken up by the calls!).

- The timing of the evaluation of 0, to calculate the overhead of the procedure calls and loop. This value is stored in the variable offset the first time the test procedure is called. The expression 0 should take a negligible amount of time, so the number of ticks timed is actually the time taken by the procedure calls and loop calculations. Subtracting this time from the other times gives a fair view of how long the expressions take, relative to one another. (Thanks to Wouter for this offset idea.)

- Use of Forbid and Permit to turn off multi-tasking temporarily, making the CPU calculate only the expressions (rather than dealing with screen output, other programs, etc.).

- Use of CtrlC and CleanUp to allow the user to stop the program if it gets too boring...

- Use of the option LARGE (using OPT) to produce an executable that uses the large data and code model. This seems to help make the timings less susceptible variations due to, for instance, optimisations, and so better for comparison. See the 'Reference Manual' for more details.

Also supplied are some example outputs. The first was from an A1200 with 2MB Chip RAM and 4MB Fast RAM. The second was from an A500Plus with 2MB Chip RAM. Both used the constant LOTS_OF_TIMES as 500,000, but you might need to increase this number to compare, for instance, an A4000/040 to an A4000/030. However, 500,000 gives a pretty long wait for results on the A500.

```
MODULE 'dos/dos'

CONST TICKS_PER_MINUTE=TICKS_PER_SECOND*60, LOTS_OF_TIMES=500000

DEF x, y, offset

PROC fred(n)
  DEF i
  i:=n+x
ENDPROC

/* Repeat evaluation of an expression */
PROC repeat(exp)
  DEF i
  FOR i:=0 TO LOTS_OF_TIMES
    Eval(exp)  /* Evaluate the expression */
  ENDFOR
ENDPROC

/* Time an expression, and set-up offset if not done already */
PROC test(exp, message)
  DEF t
  IF offset=0 THEN offset:=time(`0)  /* Calculate offset */
  t:=time(exp)
```

```
  WriteF('\s:\t\d ticks\n', message, t-offset)
ENDPROC

/* Time the repeated calls, and calculate number of ticks */
PROC time(x)
  DEF ds1:datestamp, ds2:datestamp
  Forbid()
  DateStamp(ds1)
  repeat(x)
  DateStamp(ds2)
  Permit()
  IF CtrlC() THEN CleanUp(1)
ENDPROC ((ds2.minute-ds1.minute)*TICKS_PER_MINUTE)+ds2.tick-ds1.tick

PROC main()
  x:=9999
  y:=1717
  test(`x+y,     'Addition')
  test(`y-x,     'Subtraction')
  test(`x*y,     'Multiplication')
  test(`x/y,     'Division')
  test(`x OR y,  'Bitwise OR')
  test(`x AND y, 'Bitwise AND')
  test(`x=y,     'Equality')
  test(`x<y,     'Less than')
  test(`x<=y,    'Less than or equal')
  test(`y:=1,    'Assignment of 1')
  test(`y:=x,    'Assignment of x')
  test(`y++,     'Increment')
  test(`IF FALSE THEN y ELSE x, 'IF FALSE')
  test(`IF TRUE THEN y ELSE x,  'IF TRUE')
  test(`IF x THEN y ELSE x,     'IF x')
  test(`fred(2),  'fred(2)')
ENDPROC
```

Here's the output from the A1200:

```
Addition:  22 ticks
Subtraction: 22 ticks
Multiplication:  69 ticks
Division:  123 ticks
Bitwise OR:  33 ticks
Bitwise AND: 27 ticks
Equality:  44 ticks
Less than: 43 ticks
Less than or equal:  70 ticks
Assignment of 1: 9 ticks
Assignment of x: 38 ticks
Increment: 23 ticks
IF FALSE:  27 ticks
IF TRUE: 38 ticks
IF x:  44 ticks
fred(2): 121 ticks
```

Compare this to the output from the A500Plus:

```
Addition:  118 ticks
```

```
     Subtraction: 117 ticks
     Multiplication:  297 ticks
     Division:  643 ticks
     Bitwise OR:  118 ticks
     Bitwise AND: 117 ticks
     Equality:  164 ticks
     Less than: 164 ticks
     Less than or equal:  164 ticks
     Assignment of 1: 60 ticks
     Assignment of x: 102 ticks
     Increment: 134 ticks
     IF FALSE:  118 ticks
     IF TRUE: 164 ticks
     IF x:  193 ticks
     fred(2): 523 ticks
```

Evidence, if it were needed, that the A1200 is roughly five times faster than an A500, and that’s not using the special 68020 CPU instructions!

1.162 beginner.guide/Argument Parsing

Argument Parsing
****

There are two examples in this chapter. One is for any AmigaDOS and the other is for AmigaDOS 2.0 and above. They both illustrate how to parse the arguments to your program. If your program is started from the Shell/CLI the arguments follow the command name on the command line, but if it was started from Workbench (i.e., you double-clicked on an icon for the program) then the arguments are those icons that were also selected at that time (see your Workbench manual for more details).

Any AmigaDOS

AmigaDOS 2.0 (and above)

1.163 beginner.guide/Any AmigaDOS

Any AmigaDOS
============

This first example works with any AmigaDOS. The first thing that is done is the assignment of wbmessage to a correctly typed pointer. At the same time we can check to see if it is NIL (i.e., whether the program was started from Workbench or not). If it was not started from Workbench the arguments in arg are printed. Otherwise we need to use the fact that wbmessage is really a pointer to a wbstartup object (defined in module workbench/startup), so we can get at the argument list. Then for each argument in the list we need to check the lock supplied with the argument. If it's a proper lock it will be a lock on the directory containing the argument file. The name in the argument is just a filename, not a complete path, so to read the file we need to change the current directory to the lock directory. Once we've got a valid lock and we've changed directory to there, we can find the length of the file (using FileLength) and print it. If there was no lock or the file did not exist, the name of the file and an appropriate error message is printed.

```
MODULE 'workbench/startup'

PROC main()
  DEF startup:PTR TO wbstartup, args:PTR TO wbarg, i, oldlock, len
  IF (startup:=wbmessage)=NIL
    WriteF('Started from Shell/CLI\n   Arguments: "\s"\n', arg)
  ELSE
    WriteF('Started from Workbench\n')
    args:=startup.arglist
    FOR i:=1 TO startup.numargs  /* Loop through the arguments */
      IF args[].lock=NIL
        WriteF('  Argument \d: "\s" (no lock)\n', i, args[].name)
      ELSE
        oldlock:=CurrentDir(args[].lock)
        len:=FileLength(args[].name)  /* Do something with file */
        IF len=-1
          WriteF('  Argument \d: "\s" (file does not exist)\n',
                 i, args[].name)
        ELSE
          WriteF('  Argument \d: "\s", file length is \d bytes\n',
                 i, args[].name, len)
        ENDIF
        CurrentDir(oldlock) /* Important: restore current dir */
      ENDIF
      args++
    ENDFOR
  ENDIF
ENDPROC
```

When you run this program you'll notice a slight difference between arg and the Workbench message: arg does not contain the program name, just the arguments, whereas the first argument in the Workbench argument list is the program. You can simply ignore the first Workbench argument in the list if you want.

1.164 beginner.guide/AmigaDOS 2.0 (and above)

AmigaDOS 2.0 (and above)
========================

This second program can be used as the Shell/CLI part of the previous program to provide much better command line parsing. It can only be used with AmigaDOS 2.0 and above (i.e., OSVERSION which is 37 or more). The template FILE/M used with ReadArgs gives command line parsing similar to

C’s argv array. The template can be much more interesting than this, but for more details you need the ‘AmigaDOS Manual’.

```
OPT OSVERSION=37

PROC main()
  DEF templ, rdargs, args=NIL:PTR TO LONG, i
  IF wbmessage=NIL
    WriteF('Started from Shell/CLI\n')
    templ:='FILE/M'
    rdargs:=ReadArgs(templ,{args},NIL)
    IF rdargs
      IF args
        i:=0
        WHILE args[i]  /* Loop through arguments */
          WriteF('   Argument \d: "\s"\n', i, args[i])
          i++
        ENDWHILE
      ENDIF
      FreeArgs(rdargs)
    ENDIF
  ENDIF
ENDPROC
```

As you can see the result of the ReadArgs call with this template is an array of filenames. The special quoting of filenames is dealt with correctly (i.e., when you use " around a filename that contains spaces). You need to do all this kind of work yourself if you use the arg method.

1.165 beginner.guide/Gadgets IDCMP and Graphics

Gadgets, IDCMP and Graphics
****

There are three examples in this chapter. The first shows how to open a window and put some gadgets on it. The second shows how to decipher Intuition messages that arrive via IDCMP. The third draws things with the graphics functions.

1.166 beginner.guide/Gadgets

```
Gadgets
=======
```

The following program illustrates how to create a gadget list and use it:

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = 4 * GADGETSIZE

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, next, wptr
  next:=Gadget(buf,  NIL, 1, 0, 10, 30, 50, 'Hello')
  next:=Gadget(next, buf, 2, 3, 70, 30, 50, 'World')
  next:=Gadget(next, buf, 3, 1, 10, 50, 50, 'from')
  next:=Gadget(next, buf, 4, 0, 70, 50, 70, 'gadgets')
  wptr:=OpenW(20,50,200,100, 0, WFLG_ACTIVATE,
              'Gadgets in a window',NIL,1,buf)
  IF wptr         /* Check to see we opened a window */
    Delay(500)    /* Wait a bit */
    CloseW(wptr)  /* Close the window */
  ELSE
    WriteF('Error -- could not open window!')
  ENDIF
ENDPROC
```

Four gadgets are created using an appropriately sized array as the buffer. These gadgets are passed to OpenW (the last parameter). If the window could be opened a small delay is used so that the window is visible before the program closes it and terminates. Delay is an Amiga system function from the DOS library, and Delay(n) waits n/50 seconds. Therefore, the window stays up for 10 seconds, which is enough time to play with the gadgets and see what the different types are. The next example will show a better way of deciding when to terminate the program (using the standard close gadget).

1.167 beginner.guide/IDCMP Messages

```
                IDCMP Messages
==============
```

This next program shows how to use WaitIMessage with a gadget.

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = GADGETSIZE, OURGADGET = 1

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, wptr, class, gad:PTR TO gadget
  Gadget(buf, NIL, OURGADGET, 1, 10, 30, 100, 'Press Me')
  wptr:=OpenW(20,50,200,100,
              IDCMP_CLOSEWINDOW OR IDCMP_GADGETUP,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
```

```
                 'Gadget message window',NIL,1,buf)
  IF wptr              /* Check to see we opened a window */
    WHILE (class:=WaitIMessage(wptr))<>IDCMP_CLOSEWINDOW
      gad:=MsgIaddr()  /* Our gadget clicked? */
      IF (class=IDCMP_GADGETUP) AND (gad.userdata=OURGADGET)
        TextF(10,60,
              IF gad.flags=0 THEN 'Gadget off ' ELSE 'Gadget on   ')
      ENDIF
    ENDWHILE
    CloseW(wptr)       /* Close the window */
  ELSE
    WriteF('Error -- could not open window!')
  ENDIF
ENDPROC
```

The gadget reports its state when you click on it, using the TextF function (see Graphics functions). The only way to quit the program is using the close gadget of the window. The gadget object is defined in the module intuition/intuition and the iaddr part of the IDCMP message is a pointer to our gadget if the message was a gadget message. The userdata element of the gadget identifies the gadget that was clicked, and the flags element is zero if the boolean gadget is off (unselected) or non-zero if the boolean gadget is on (selected).

1.168 beginner.guide/Graphics

Graphics
========

The following program illustrates how to use the various graphics functions.

```
MODULE 'intuition/intuition'

PROC main()
  DEF wptr, i
  wptr:=OpenW(20,50,200,100,IDCMP_CLOSEWINDOW,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Graphics demo window',NIL,1,NIL)
  IF wptr  /* Check to see we opened a window */
    Colour(1,3)
    TextF(20,30,'Hello World')
    SetTopaz(11)
    TextF(20,60,'Hello World')
    FOR i:=10 TO 150 STEP 8  /* Plot a few points */
      Plot(i,40,2)
    ENDFOR
    Line(160,40,160,70,3)
    Line(160,70,170,40,2)
    Box(10,75,160,85,1)
    WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
```

```
    ENDWHILE
    CloseW(wptr)
  ELSE
    WriteF('Error -- could not open window!\n')
  ENDIF
ENDPROC
```

First of all a small window is opened with a close gadget and activated (so it is the selected window). Clicks on the close gadget will be reported via IDCMP, and this is the only way to quit the program. The graphics functions are used as follows:

* Colour is used to set the foreground colour to pen one and the background colour to pen three. This will make the text nicely highlighted.

* Text is output in the standard font.

* The font is set to Topaz 11.

* More text is output (probably now in a different font).

* The FOR loop plots a dotted line in pen two.

* A vertical line in pen three is drawn.

* A diagonal line in pen two is drawn. This and the previous line together produce a vee shape.

* A filled box is drawn in pen one.

1.169 beginner.guide/Screens

Screens
=======

This next example uses parts of the previous example, but also opens a custom screen. Basically, it draws coloured lines and boxes in a big window opened on a 16 colour, high resolution screen.

```
MODULE 'intuition/intuition', 'graphics/view'

PROC main()
  DEF sptr=NIL, wptr=NIL, i
  sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
  IF sptr
    wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
                WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
                'Graphics demo window',sptr,$F,NIL)
    IF wptr
      TextF(20,20,'Hello World')
      FOR i:=0 TO 15  /* Draw a line and box in each colour */
        Line(20,30,620,30+(7*i),i)
        Box(10+(40*i),140,30+(40*i),170,1)
```

```
            Box(11+(40*i),141,29+(40*i),169,i)
          ENDFOR
          WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
          ENDWHILE
          WriteF('Program finished successfully\n')
        ELSE
          WriteF('Could not open window\n')
        ENDIF
      ELSE
        WriteF('Could not open screen\n')
      ENDIF
      IF wptr THEN CloseW(wptr)
      IF sptr THEN CloseS(sptr)
    ENDPROC
```

As you can see, the error-checking IF blocks can make the program hard to read. Here's the same example written with an exception handler:

```
    MODULE 'intuition/intuition', 'graphics/view'

    ENUM WIN=1, SCRN

    RAISE WIN  IF OpenW()=NIL,
          SCRN IF OpenS()=NIL

    PROC main() HANDLE
      DEF sptr=NIL, wptr=NIL, i
      sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
      wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
                  WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
                  'Graphics demo window',sptr,$F,NIL)
      TextF(20,20,'Hello World')
      FOR i:=0 TO 15  /* Draw a line and box in each colour */
        Line(20,30,620,30+(7*i),i)
        Box(10+(40*i),140,30+(40*i),170,1)
        Box(11+(40*i),141,29+(40*i),169,i)
      ENDFOR
      WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
      ENDWHILE
    EXCEPT DO
      IF wptr THEN CloseW(wptr)
      IF sptr THEN CloseS(sptr)
      SELECT exception
      CASE 0
        WriteF('Program finished successfully\n')
      CASE WIN
        WriteF('Could not open window\n')
      CASE SCRN
        WriteF('Could not open screen\n')
      ENDSELECT
    ENDPROC
```

It's much easier to see what's going on here. The real part of the program (the bit before the EXCEPT) is no longer cluttered with error checking, and it's easy to see what happens if an error occurs. Notice that if the program successfully finishes it still has to close the screen and window properly, so it's often sensible to use EXCEPT DO to raise a zero exception and deal with all the tidying up in the handler.

1.170 beginner.guide/Recursion Example

Recursion Example
****

This next example uses a pair of mutually recursive procedures to draw what is known as a dragon curve (a pretty, space-filling pattern).

```
MODULE 'intuition/intuition', 'graphics/view'

/* Screen size, use SIZEY=512 for a PAL screen */
CONST SIZEX=640, SIZEY=400

/* Exception values */
ENUM WIN=1, SCRN, STK, BRK

/* Directions (DIRECTIONS gives number of directions) */
ENUM NORTH, EAST, SOUTH, WEST, DIRECTIONS

RAISE WIN  IF OpenW()=NIL,
      SCRN IF OpenS()=NIL

/* Start off pointing WEST */
DEF state=WEST, x, y, t

/* Face left */
PROC left()
  state:=Mod(state-1+DIRECTIONS, DIRECTIONS)
ENDPROC

/* Move right, changing the state */
PROC right()
  state:=Mod(state+1, DIRECTIONS)
ENDPROC

/* Move in the direction we're facing */
PROC move()
  SELECT state
  CASE NORTH; draw(0,t)
  CASE EAST;  draw(t,0)
  CASE SOUTH; draw(0,-t)
  CASE WEST;  draw(-t,0)
  ENDSELECT
ENDPROC

/* Draw and move to specified relative position */
PROC draw(dx, dy)
  /* Check the line will be drawn within the window bounds */
  IF (x>=Abs(dx)) AND (x<=SIZEX-Abs(dx)) AND
     (y>=Abs(dy)) AND (y<=SIZEY-10-Abs(dy))
    Line(x, y, x+dx, y+dy, 2)
  ENDIF
```

```
    x:=x+dx
    y:=y+dy
  ENDPROC

  PROC main() HANDLE
    DEF sptr=NIL, wptr=NIL, i, m
    /* Read arguments:        [m [t [x  [y]]]] */
    /* so you can say: dragon  16              */
    /*             or: dragon  16 1            */
    /*             or: dragon  16 1 450        */
    /*             or: dragon  16 1 450 100    */
    /* m is depth of dragon, t is length of lines */
    /* (x,y) is the start position */
    m:=Val(arg, {i})
    t:=Val(arg:=arg+i, {i})
    x:=Val(arg:=arg+i, {i})
    y:=Val(arg:=arg+i, {i})
    /* If m or t is zero use a more sensible default */
    IF m=0 THEN m:=5
    IF t=0 THEN t:=5
    sptr:=OpenS(SIZEX,SIZEY,4,V_HIRES OR V_LACE,'Dragon Curve Screen')
    wptr:=OpenW(0,10,SIZEX,SIZEY-10,
                IDCMP_CLOSEWINDOW,WFLG_CLOSEGADGET,
                'Dragon Curve Window',sptr,$F,NIL)
    /* Draw the dragon curve */
    dragon(m)
    WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
    ENDWHILE
  EXCEPT DO
    IF wptr THEN CloseW(wptr)
    IF sptr THEN CloseS(sptr)
    SELECT exception
    CASE 0
      WriteF('Program finished successfully\n')
    CASE WIN
      WriteF('Could not open window\n')
    CASE SCRN
      WriteF('Could not open screen\n')
    CASE STK
      WriteF('Ran out of stack in recursion\n')
    CASE BRK
      WriteF('User aborted\n')
    ENDSELECT
  ENDPROC

  /* Draw the dragon curve (with left) */
  PROC dragon(m)
    /* Check stack and ctrl-C before recursing */
    IF FreeStack()<1000 THEN Raise(STK)
    IF CtrlC() THEN Raise(BRK)
    IF m>0
      dragon(m-1)
      left()
      nogard(m-1)
    ELSE
      move()
    ENDIF
```

```
ENDPROC

/* Draw the dragon curve (with right) */
PROC nogard(m)
  IF m>0
    dragon(m-1)
    right()
    nogard(m-1)
  ELSE
    move()
  ENDIF
ENDPROC
```

If you write this to the file dragon.e and compile it to the executable dragon then some good things to try are:

```
dragon 5 9 300 100
dragon 10 4 250 250
dragon 11 3 250 250
dragon 15 1 300 100
dragon 16 1 400 150
```

If you want to understand how the program works you need to study the recursive parts. Here's an overview of the program, outlining the important aspects:

- The constants SIZEX and SIZEY are the width and height (respectively) of the custom screen (and window). As the comment suggests, change SIZEY to 512 if you want a bigger screen and you have a PAL Amiga.
- The state variable holds the current direction (north, south, east or west).
- The left and right procedures turn the current direction to the left and right (respectively) by using some modulo arithmetic trickery.
- The move procedure uses the draw procedure to draw a line (of length t) in the current direction from the current point (stored in x and y).
- The draw procedure draws a line relative to the current point, but only if it fits within the boundaries of the window. The current point is moved to the end of the line (even if it isn't drawn).
- The main procedure reads the command line arguments into the variables m, t, x and y. The depth/size of the dragon is given by m (the first argument) and the length of each line making up the dragon is given by t (the second argument). The starting point is given by x and y (the final two arguments). The defaults are five for m and t, and zero for x and y.
- The main procedure also opens the screen and window, and sets the dragon drawing.
- The dragon and nogard procedures are very similar, and these are responsible for creating the dragon curve by calling the left, right and move procedures.

- The dragon procedure contains a couple of checks to see if the user has pressed Control-C or if the program has run out of stack space, raising an appropriate exception if necessary. These exceptions are handled by the main procedure.

Notice the use of Val and the exception handling. Also, the important base case of the recursion is when m reaches zero (or becomes negative, but that shouldn't happen). If you start off a big dragon and want to stop it you can press Control-C and the program tidies up nicely. If it has finished drawing you simply click the close gadget on the window.

1.171 beginner.guide/Common Problems

Common Problems
****

If you are new to programming or the Amiga E language then you might appreciate some help locating problems (or bugs) in your programs. This Appendix details some of the most common mistakes people make.

1.172 beginner.guide/Assignment and Copying

Assignment and Copying
======================

This is probably the most common problem encountered by people who are used to languages like BASIC. Strings, lists, arrays and objects cannot be initialised using an assignment statement: data must be copied. Unlike BASIC, this kind of data is represented by a pointer (see
PTR Type
), so

only the pointer would be copied by an assignment statement, not the data it points to. The following examples all copy a pointer rather than the data, and so the memory for the data is shared (and this is probably not what was intended).

```
       DEF s[30]:STRING, t[30]:STRING,
           l[10]:LIST, m[10]:LIST,
           x:myobj, y:myobj,
           a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

     /* You probably don't want to do any of these */
       s:='Some text in a string'
       l:=[-6,4,-9]
       x:=[1,2,3]:myobj
       a:=[1,-3,8,7]:INT

       t:=s
       m:=l
       y:=x
       b:=a
```

All the declarations allocate memory for the appropriate data. The first four assignments replace the pointers to this memory with pointers to some statically allocated memory. The memory allocated by the declarations is probably now unreachable, because the only pointers to it have been over-written. BASIC programmers might expect, say, the assignment to s to have copied the string into the memory allocated for s by its declaration, but this is not the case (only the pointer to the string is copied).

For the E-string, s, and E-list, l, there is another, disastrous side-effect. The assignment to s, for example, means that s will point to a normal string, not an E-string. So, s can no longer be used with any of the E-string functions. The same applies to the E-list, l.

The final four assignments also copy only the pointers. This means that s and t will point to exactly the same memory. So they will represent exactly the same string, and any change to one of them (by a StrAdd, for example) will appear to change both (of course, only one lump of memory is being changed, but there are two references to it). This is called memory sharing, and is only a problem if you didn't intend to do it!

To get the result that a BASIC programmer might have intended you need to copy the appropriate data. For E-strings and E-lists the functions to use are, respectively, StrCopy and ListCopy. All other data must be copied using a function like CopyMem (an Amiga system function from the Exec library). (Normal strings can be copied using AstrCopy built-in function, see the 'Reference Manual'.) Here's the revised forms of the above assignments:

```
       DEF s[30]:STRING, t[30]:STRING,
           l[10]:LIST, m[10]:LIST,
           x:myobj, y:myobj,
           a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

       StrCopy(s, 'Some text in a string')  /* Defaults to ALL */
       ListCopy(l, [-6,4,-9])               /* Defaults to ALL */
```

```
CopyMem([1,2,3]:myobj, x, SIZEOF myobj)
CopyMem([1,-3,8,7]:INT, a, 4*SIZEOF INT)

StrCopy(t, s)   /* Defaults to ALL */
ListCopy(m, l)  /* Defaults to ALL */
CopyMem(x, y, SIZEOF myobj)
CopyMem(a, b, 4*SIZEOF INT)
```

Notice that you need to supply the size (in bytes) of the data being copied when you use CopyMem. The parameters are also given in a slightly different order to the E-string and E-list copying functions (i.e., the source must be the first parameter and the destination the second). The CopyMem function does a byte-by-byte copy, something like this:

```
PROC copymem(src, dest, size)
  DEF i
  FOR i:=1 TO size DO dest[]++:=src[]++
ENDPROC
```

Of course, you can use string constants and lists to give initialised arrays, but in this case you should be initialising an appropriately typed pointer. You must also be careful not to run into a static data problem (see Static data).

```
DEF s:PTR TO CHAR, l:PTR TO LONG, x:PTR TO myobj, a:PTR TO INT
s:='Some text in a string'
l:=[-6,4,-9]
x:=[1,2,3]:myobj
a:=[1,-3,8,7]:INT
```

1.173 beginner.guide/Pointers and Memory Allocation

Pointers and Memory Allocation
==============================

Another common error is to declare a pointer (usually a pointer to an object) and then use it without the memory for the target data being allocated.

```
/* You don't want to do this */
  DEF p:PTR TO object
  p.element:=99
```

There are two ways of correcting this: either dynamically allocate the memory using NEW or, more simply, let an appropriate declaration allocate it. See Memory Allocation .

```
DEF p:PTR TO object
NEW p
```

```
       p.element:=99

       DEF p:object
       p.element:=99
```

1.174 beginner.guide/String and List Misuse

String and List Misuse
======================

Some of the string functions can only be used with E-strings. Generally, these are the ones that might extend the string. If you use a normal string instead you can run into some serious (but subtle) problems. Commonly misused functions are ReadStr, MidStr and RightStr. Similar problems can arise by using a list when an E-list is required by a list function.

String constants and normal lists are static data, so you shouldn't try to alter their contents unless you know what you're doing (see

Static data
).

1.175 beginner.guide/Initialising Data

Initialising Data
=================

Probably one of the most common mistakes that even seasoned programmers make is to forget to initialise variables (especially pointers). The rules in the 'Reference Manual' state which declarations initialise variables to zero values, but it is often wise to make even these explicit (using initialised declarations). Variable initialisation becomes even more important when using automatic exceptions.

1.176 beginner.guide/Freeing Resources

Freeing Resources
=================

Unlike a Unix operating system, the Amiga operating system requires the programmer to release or free any resources used by a program. In practice, this means that all windows, screens, libraries, etc., that are successfully opened must be closed before the program terminates. Amiga E provides some help, though: the four most commonly used libraries (Dos, Exec, Graphics and Intuition) are opened before the start of an E program and closed at the end (or when CleanUp is called). Also, memory allocated using New, List and String is automatically freed at the end of a program.

1.177 beginner.guide/Pointers and Dereferencing

Pointers and Dereferencing
==========================

C programmers may think that the ^var and {var } expressions are the direct equivalent of C's &var and *var expressions. However, in E dereferencing is normally achieved using array and object element selection, and pointers to large amounts of data (like E-strings or objects) are made by declarations. This means that the ^var and {var } expressions are rarely used, whilst var[] is very common.

1.178 beginner.guide/Mathematics Functions

Mathematics Functions
=====================

The standard mathematical operators / and * do not use full 32-bit values in their calculations, as noted previously (see

Maths and logic functions

). A common problem is to forget this and use them where the values will exceed the 16-bit limit. A typical example is the position calculations used with proportional gadgets. See

Signed and Unsigned Values

.

1.179 beginner.guide/Signed and Unsigned Values

Signed and Unsigned Values
==========================

This is a quite advanced topic, but might be the cause of some strange bugs in your programs. Basically, E does not have a way of differentiating signed and unsigned values from, say, the LONG type. That is, all values from the 32-bit, LONG type are considered to be signed values, so the range of values is from -2,147,483,648 to 2,147,483,647. If the values from this type were taken to be unsigned then no negative values would be allowed but more positive values would be possible (i.e., the range of values would be from zero to 4,294,967,295). This distinction would also affect the mathematical operators.

In practice, though, it is not the LONG type that can cause problems. Instead, it is the 16-bit, INT type, which again is considered to be signed. This means that the range of values is -32,768 to 32,767. However, the Amiga system objects contain a number of 16-bit, INT elements which are actually interpreted as unsigned, ranging from zero to 65,535. A prominent example is the proportional gadget which forms a part of a scroll-bar on a window (for example, a drawer window on Workbench). This works with unsigned 16-bit values, which is at odds with the INT type in E. These values are commonly used in calculations to determine the position of something displayed in a window, and if the INT type is used without taking into account this signed/unsigned problem the results can be quite wrong. Luckily it is quite simple to convert the signed INT values into unsigned values if they are part of some expression, since the value of any expression is taken from the LONG type (and unsigned INT values fit well within the range of even signed LONG values).

```
     PROC unsigned_int(x) IS x AND $FFFF
```

The function unsigned_int, above, is very specific to the way the Amiga handles values internally, so to understand how it works is beyond the scope of this Guide. It should be used wherever an unsigned 16-bit value is stored in an INT element of, say, an Amiga system object. For example, the position of the top of a (vertical) proportional gadget as a percentage (zero to one hundred) of its size can be calculated like this:

```
       /* propinfo is from the module 'intuition/intuition' */
       DEF gad:PTR TO propinfo, pct
       /* Set up gad... */
       /* Calculate percentage (MAXPOT is from 'intuition/intuition') */
       pct:=Div(Mul(100,unsigned_int(gad.vertpot)),MAXPOT)
```

Notice that the full 32-bit functions Div and Mul need to be used since the arithmetic may be well over the normal 16-bits used in the / and * operators.

The remaining type, CHAR, is not, in practice, a problem. It is the only unsigned type, with a range of values from zero to 255. There is a fairly simple way to convert these values to signed values (and again this is particular to the way the Amiga stores values internally). One good example of a signed CHAR value is the priority value associated with a node of an Amiga list (i.e., the pri element of an ln object from the module exec/nodes).

```
     PROC signed_char(x) IS IF x<128 THEN x ELSE x-256
```

1.180 beginner.guide/Other Information

```
                Other Information
*****************
```

This Appendix contains some useful, miscellaneous information.

1.181 beginner.guide/Amiga E Versions

Amiga E Versions
================

As I write, the current version of Amiga E is version 3.1a (which is major update of v3.0e). This edition of the Guide is based primarily on that version, but the majority still applies to the older versions, including the last Public Domain version (v2.1b). Version 3.2 is imminent, and this Guide is hopefully included with this major update. See the 'Reference Manual' for details of the new features and changes.

Please note that, as of v3.0a, Amiga E is a commercial product so you must pay a fee to get a version of the full compiler (which will be registered to you). The Public Domain distribution contains only a demonstration version of the compiler, with limited functionality. See the 'Reference Manual' for more details.

1.182 beginner.guide/Further Reading

Further Reading
===============

'Amiga E Language Reference'
Referred to as the 'Reference Manual' in this Guide. This is one of the documents that comes with the Amiga E package, and is essential reading since it was written by Wouter (the author of Amiga E). It contains a lot of extra information.

'Rom Kernel Reference Manual' (Addison-Wesley)
This is the official Commodore documentation on the Amiga system functions and is a must if you want to use these functions properly. At the time of writing the Third Edition is the most current and it covers the Amiga system functions up to Release 2 (i.e., AmigaDOS 2.04 and KickStart 37). Because there is so much information it comes in three separate volumes: 'Libraries', 'Includes and Autodocs', and 'Devices'. The 'Libraries' volume is probably the most useful as it contains many examples and a lot of tutorial material. However, the examples are written mainly in C (the remainder are in Assembly). To alleviate this problem I have

undertaken to re-code them in E, and Part One of this effort should be available from the same place you got this Guide (the archive name will be something like JRH-RKRM-1).

'The AmigaDOS Manual' (Bantam Books)
This is the companion to the 'Rom Kernel Reference Manual' and is the official Commodore book on AmigaDOS (both the AmigaDOS programs and the DOS library functions). Again, the Third Edition is the most current.

Example sources
Amiga E comes with a large collection of example programs. When you're familiar with the language you should be able to learn quite a bit from these. There are a lot of small, tutorial programs and a few large, complicated programs.

1.183 beginner.guide/Amiga E Author

Amiga E Author

In case you didn't know the author and creator of Amiga E is Wouter van Oortmerssen (or $#%!). You can reach him by normal mail at the following address:

Wouter van Oortmerssen ($#%!)
Levendaal 87
2311 JG Leiden
HOLLAND

However, he much prefers to chat by E-mail, and you can reach him at the following addresses:

Wouter@alf.let.uva.nl (E-programming support)
Wouter@mars.let.uva.nl (personal)
Oortmers@gene.fwi.uva.nl (other)

Better still, if your problem or enquiry is of general interest to Amiga E users you may find it useful joining the Amiga E mailing list. Wouter regularly contributes to this list and there are a number of good programmers who are at hand to help or discuss problems. To join send a message to:

amigae-request@bkhouse.cts.com

Once you're subscribed, you will receive a copy of each message mailed to the list. You will also receive a message telling you how you can contribute (i.e., ask questions!).

1.184 beginner.guide/Guide Author

Guide Author

This Guide was written by Jason Hulance, with a lot of help and guidance from Wouter. The original aim was to produce something that might be a useful introduction to Amiga E for beginners, so that the language could (rightly) become more widespread. The hidden agenda was to free Wouter from such a task so that he could concentrate his efforts on improving Amiga E.

You can reach me by normal mail most easily at the following (work) address:

Jason R. Hulance
Formal Systems (Europe) Ltd.
3 Alfred Street
Oxford
OX1 4EH
ENGLAND

Alternatively, you can find me on the Amiga E mailing list, or E-mail me directly at one of the following addresses:

jason@fsel.com
m88jrh@uk.ac.oxford.ecs

If you have any changes or additions you'd like to see then I'd be very happy to consider them. Criticism of the text is also welcome, especially if you can suggest a better way of explaining things. I am also keen to hear from people who can highlight areas that are particularly confusing or badly worded!

Also, for a small fee you get a printable version of this Guide, in either DVI or PostScript form. This includes a huge index, several pictures and many nice tables, and costs only 5 pounds for UK residents, or 8 pounds for non-UK residents (prices include disk and postage costs). I can also make printed versions (including proper binding if required) for an extra cost. Please feel free to E-mail or write to me at the above addresses if you'd like more details.

1.185 beginner.guide/E Language Index

E Language Index

This index should be used to find detailed information about the keywords, functions, variables and constants which are part of the Amiga E language. There is a separate index which deals with concepts etc. (see

Main Index
).

1.186 beginner.guide/Main Index

Main Index
****

This index should be used to find detailed information about particular concepts. There is a separate index which deals with the keywords, variables, functions and constants which are part of Amiga E (see

E Language Index
).