Borland C++ menu commands

See also

The Borland C++ IDE (integrated development environment) offers everything you need to write, edit, compile, link, run, manage, and debug your programs. The menu bar at the top of the window is the gateway to the functionality of the IDE.

To go to the menu bar, press F10 or click anywhere on it.

You can choose any of the following commands on the menu bar:

File menu

Edit menu

Search menu

View menu

Project menu

Script Menu

Tool menu

Debug menu

Options menu

Window menu

Help menu

In addition to the menu bar, you can choose commands from the Control menu located in the upper left corner of the active window.

Shortcut: Help for menu commands is also available by highlighting the menu command and pressing F1.

How to use menus

You can select a menu command the following ways:

- Click the title of the menu you want to pull down, drag the mouse pointer over the menu command you want to choose, and then release the mouse button.
- Click the title of the menu you want to pull down and then click the menu command you want to choose.
- Press F10, then use the arrow keys to go to the menu you want and use them again to select a command. Press Enter to choose the selected command.
- Press Alt and the underlined letter of the menu you want (such as Alt+F for the File menu), use the arrow keys to select a command, then press Enter to choose that command. You can also press the underlined letter of a menu name or command instead of using the arrow keys.

▶ Menu commands can be followed by either an ellipsis mark (...), an arrow, or a hot key:

- The ellipsis displays a dialog box
- The arrow displays another menu
- The hot key initiates the command when pressed

Choose this button to save the selections you made.
If OK is the default button, you can just press Enter to choose it.

The Undo Page button restores the current screen of options to the settings present when you selected the Options Settings dialog box.

Choose this button to close the dialog box and discard your changes. No changes are saved and no action occurs. Esc is always a keyboard shortcut for Cancel even if a Cancel button is not available.

Choose this button to close the dialog box with the current settings.

Esc is always a keyboard shortcut for Close even if a Close button is not available.

Choose the Help button to get Help about the active window or dialog box.

Choose this button to connect to a network drive, if available.

Choose OK to put this Message window away.

Choose either:
Explorer style (Windows 95 and NT users)
Windows 3.1 style (NT users only)

Load dialog boxes (Explorer)

A Load dialog box displays when you choose any of the following menu commands.

Command	Load Dialog Box
File|New|AppExpert	New AppExpert Project
File|Open	Open a File
File|Save	Save File As (if a new file has not been saved)
File|Save As	Save File As
Project|Open	Open Project File

A period (**.**) in a file name is considered part of the name, not the extension. For example, if you enter `FOO.TXT` in the File Name box and the type selected is RC, the result is `FOO.TXT.RC`.

When you open a file, the Viewer associated with the file type tells the IDE how to display the file.

► You can open several files at once by selecting them with the Ctrl or Shift key, and then choosing Open.

Load dialog boxes (Windows 3.1 style)

A load dialog box displays when you choose any of the following menu commands.

Command	Load Dialog Box
File|New|AppExpert	New AppExpert Project
File|Open	Open a File
File|Save	Save File As (if a new file has not been saved)
File|Save As	Save File As
Project|Open	Open Project File

In a load dialog box, you can

- type a full file name in the File Name input box
- display selected files in the File Name list box by choosing a type in the List Files of Type list box. (You can also choose files by typing a partial file name with * or ? wildcards in the File Name input box and pressing Enter.)
- choose a selection from a history list of file specifications you entered earlier. (This option is available only when opening a file).
- view the contents of different directories by selecting one from the Directories list box. (You can also change directories by typing a directory in the Files Names input box and pressing Enter.)
- view the contents of different drives by selecting one in the Drives list box.
- connect to a network drive by selecting Network.

File Name	Enter the name of the file to open, or a file specification to limit the files that display in the list box.
Directories list box	Select the directory by that contains the files you want.
Viewer	Accept the default viewer or choose another tool you want to use to open the selected file type. (This option is available only when opening a file).
Drives	Select the drive that contains the files you want.
List Files of Type	Select which file types you want displayed.

File Name input box

Enter the name of a file, or supply a partial file name with ? or * wildcards to display selected files in the File Name list box.

This input box has a history list attached to it.

To select a file, choose any of the following actions:

- Type in a file name and choose OK or press Enter.
- Supply a file specification (such as a partial file name or type with * and ? wildcards) to display selected files in the File Name list box.
- Type in a directory followed by a file name or file specification.
- Click the drop-down arrow next to the input box (or Press Alt+Down arrow) to choose a file specification from the history list.

If you are using your keyboard:

- Press Alt+N to return the cursor to the File Name input box.
- Tab to the list box and use the arrow keys to select a file.

If Enable Filename Completion is on, you can type a few characters of the file and directory name and press the space bar to display files that match the typed in characters.

- If there is no match, the system will beep and place a space in the File Name input box.
- If there is one match, the directory and file name will be inserted in the File Name input box.
- If there is more than one match, a list will be displayed. You can choose the directory and file you want.

File Name

Displays the files in the current drive and directory that match the file-specification in the File Name input box.

To select a file, double-click on it.

If you are using the keyboard, tab to the File name list box and use the Up or Down arrow to reach the file you want, then press Enter.

Viewer

Sets the tool that will be used to open the selected file type. You can accept the default viewer or choose another tool that you want to use to open the selected file type.

- This option is available only when opening a file.
- To set the properties for a tool, use the Tool Options dialog box and click **Advanced** to display the Advanced Tool dialog box.

List Files of Type

Lets you select which file types you want displayed.

To select a type, click on it.

If you are using your keyboard, press Alt+T and use the arrow keys to select one of the available types, then press Enter.

Directories

Lets you change the current directory location of the files displayed.

Double-click directories in the Directories list box to change the current drive.

If you are using your keyboard

- Press Alt+D and the first directory in the Directories list box will be outlined.
- Use the arrow keys to select the directory you want to open and choose OK or press Enter.
- You can also enter a directory in the File Name input box to change the current directory.

Drives

Displays the drives (and volume labels) available to you.

To select a drive, click on it.

If you are using your keyboard:

- Press Alt+V and the first drive in the list box will be outlined.
- Use the arrow keys to select the drive you want and press Enter.

History list

If there is a Down arrow icon to the right of an input box, a history list is attached to the input box.

You use the history list to re-enter text that you have already entered.

The history list displays the last ten selections entered.

To choose an item from the history list, click the item to place it in the input box, and press Enter.

You can edit any item that you have selected from the history list.

To exit the history list without making a selection, click the Down arrow again (or press Alt+Down arrow), or click anywhere in the desktop outside of the input box and history list.

GREP-like regular expressions

The symbols you can use to produce GREP-like regular expressions include:

- **^** A circumflex at the start of the string matches the start of a line.
- **$** A dollar sign at the end of the expression matches the end of a line.
- **.** A period matches any character.
- ***** An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, `bo*` matches `bot`, `boo`, and as well as `bo`.
- **+** A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, `bo+` matches `bot` and `boo`, but not `b` or `bo`.
- **{ }** Characters or expressions in braces are grouped so that the evaluation of a search pattern can be controlled and so grouped text can be referred to by number.
- **[ ]** Characters in brackets match any one character that appears in the brackets, but no others. For example `[bot]` matches `b`, `o`, or `t`.
- **[^]** A circumflex at the start of the string in brackets means NOT. Hence, `[^bot]` matches any characters except `b`, `o`, or `t`.
- **[-]** A hyphen within the brackets signifies a range of characters. For example, `[b-o]` matches any character from `b` through `o`.
- **** A backslash before a wildcard character tells the C++ IDE to treat that character literally, not as a wildcard. For example, **\\^** matches ^ and does not look for the start of a line.

Brief regular expressions

The symbols you can use to produce Brief regular expressions include:

- **<** A less than at the start of the string matches the start of a line.
- **%** A percent sign at the start of the string matches the start of a line.
- **$** A dollar sign at the end of the expression matches the end of a line.
- **>** A greater than at the end of the expression matches the end of a line.
- **?** A question mark matches any single character.
- **@** An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, `bo@` matches `bot`, `boo`, and as well as `bo`.
- **+** A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, `bo+` matches `bot` and `boo`, but not `b` or `bo`.
- **|** A vertical bar matches either expression on either side of the vertical bar. For example, `bar|car` will match either `bar` or `car`.
- **~** A tilde matches any single character that is **not** a member of a set.
- **[ ]** Characters in brackets match any one character that appears in the brackets, but no others. For example `[bot]` matches `b`, `o`, or `t`.
- **[^]** A circumflex at the start of the string in brackets means NOT. Hence, `[^bot]` matches any characters except `b`, `o`, or `t`.
- **[-]** A hyphen within the brackets signifies a range of characters. For example, `[b-o]` matches any character from `b` through `o`.
- **{ }** Braces group characters or expressions. Groups can be nested, with the maximum number of ten groups in a single pattern.
- **** A backslash before a wildcard character tells the Borland C++ IDE to treat that character literally, not as a wildcard. For example, **\^** matches ^ and does not look for the start of a line.

Go to Line Number Dialog Box

Use this dialog box to find a specific line number in your source file.

Enter New Line Number
Specify which line of source code you want to go to.

Enter a specific line number (or use the history list to select a number previously entered), then press Enter (or choose OK).

Note: The current line number displays in the status line at the bottom of the IDE desktop.

Sorry, No Help is Available for that Term

You have attempted to get Language Help in an Edit window for a term that the Help system does not recognize.

SpeedMenu
Press the right mouse button to display (when available) a pop-up menu of available commands.

Expand

To show hidden subtopics associated with a topic.

A plus (**+**) sign next to a topic indicates that you can expand it in one of the following ways:

- click the **+**
- double-click the topic
- highlight the topic and press the Spacebar
- use the SpeedMenu commands

When expanded, the **+** changes to a minus (**-**) sign, and a list of subtopics displays under the topic.

Collapse

To hide subtopics displayed under a topic.

A minus (**-**) sign next to a topic indicates that you can collapse it in one of the following ways:

- click the **-**
- double-click the topic
- highlight the topic and press the Spacebar
- use the SpeedMenu commands

When a topic is collapsed, the **-** changes to a plus (**+**) sign, and the list of subtopics are hidden.

No additional information was found. This may have occurred because your installation does not include one or more of the Help files supplied with Borland C++. To install a Help file, run the Borland C++ Installation program again and choose Custom Install and then choose Online Help.

Control menu

The Control-menu box appears in the upper left corner of the active window. Click it once to display the menu. The commands on this menu affect the active window in the Borland C++ IDE.

The desktop as well as each Edit window and dialog box has its own Control menu on which one or more of the following commands are available:

Restore

Move

Size

Minimize

Maximize

Close

The following commands do not display under Windows 95.

Next

Switch to

Control | Restore

This command returns the active window to its previous size.

It is only available when the active window is maximized or minimized.

Control | Move

Use this command to move the active window with the keyboard, rather than by dragging it with the mouse.

Use the arrow keys to move the window and press Enter when you are done.

It is not available when the active window is maximized.

You can also move the window by dragging its title bar.

Control | Size

Use this command to change the size of the active window with your keyboard.

Use the arrow keys to move the window borders. Press Enter when you are satisfied with the window size.

- This command is not available if the active window is maximized or minimized.

Control | Minimize

This command turns the active window into an icon on the Borland C++ IDE desktop.

It is not available if the active window is already minimized.

Control | Maximize

This command enlarges the active window to full screen size.

It is not available if the active window is already maximized.

Control | Close

This command closes the active window or, if chosen from the desktop, closes and unloads the C++ IDE from memory.

If you have modified an Edit window and not saved the file, a Save File dialog box appears asking you if you want to save the file before closing.

- This command does not close a project, just the project view. To open the project window again, choose View|Project.
- To close a project, use either the File|Close or Project|Close command.

Control | Next

This command makes the next open window or icon active.

- Not applicable under Windows 95.

Control | Switch To

This command displays the Task List dialog box, where you can switch from one application to another and rearrange application windows.

- Not applicable under Windows 95.

"Save File?" dialog box

Asks if you want to save the file before closing.

- Choose Yes if you want to save the file before closing.
- Choose No if you do not want to save the file before closing.
- Choose Cancel to cancel the operation and return to your active Edit window.

Task List dialog box

Lets you switch to another application and rearrange application windows.

- Not applicable under Windows 95.

Task List list box
All open applications are listed in this box.

Action Buttons
Use the buttons at the bottom of the dialog box to switch to other applications and rearrange application windows.

Task List list box

Lists all open applications.

- Not applicable under Windows 95.

To choose an application using the mouse, double-click the application you want.

To choose an application using the keyboard, press the Up or Down arrow keys to highlight the application, then press Enter to choose it.

You can also use the Action Buttons at the bottom of the dialog box to switch to other applications and rearrange application windows.

Action Buttons

Action buttons switch to another application and rearrange application windows.

- With the mouse, click the action button you want.
- With the keyboard, press Tab to go to the action button, then press Enter.
- Not applicable under Windows 95.

Switch To
Switches to the application you have selected in the list box.

End Task
Closes the selected application in the list box.

Cancel
Closes the Task List dialog box.

Cascade
Arranges the application windows in an overlap style.

Tile
Arranges the application windows in a tile fashion (side-by-side).

Arrange Icons
Evenly spaces the icons across the bottom of the desktop.

File menu

The File menu provides commands for creating, opening, saving, and closing files and projects, as well as printing files and exiting the Borland C++ IDE.

New

Open

Close

Save

Save as

Save all

Print

Printer Setup

Send

Exit

List of closed files

File | New

This menu provides commands to open a file into the Borland C++ IDE:

Choose...	To...
Text Edit	open a blank Edit window.
Project	create a new project.
AppExpert	automatically generate code for an ObjectWindows-based application.
Resource Project	create a new resource project.

File | New | Text Edit

This command opens a blank Edit window and loads a file with the default name NONAMExx.CPP (where xx stands for a number). It automatically makes the new Edit window active. NONAME files are used as a temporary edit buffer and the Borland C++ IDE prompts you to supply a new name when saved. If you load a file into an active Edit window that contains an empty NONAME file, the contents of the Edit window is replaced.

File | New | Project

This command displays the **New Target** dialog box where you can create and name a new project. When a new project is created by the Project Manager, a skeleton project is created and it inherits the option settings of the project that was last opened. If you have not opened a project in your current Borland C++ session, the new project is given the settings of the Default Style Sheet.

After you create the initial target for your project, you can modify the skeleton to fit your needs. The project manager can load and use projects from previous versions of Borland C++. Once an old project is loaded into the project tree, it is converted to the new project format.

Once your project is created, you can change Target Attributes by selecting TargetExpert from the SpeedMenu on the project tree.

File | New | AppExpert

This command displays the New AppExpert Project dialog box, where you specify the name and location of the AppExpert project you want to create.

- If you choose an existing project, that project will be overwritten with a new AppExpert project.
- After you name your project, the AppExpert multi-page dialog displays, where you set options for the automatically generated AppExpert code. After setting these options, AppExpert generates an ObjectWindows-based Windows application.

File | New | Resource Project

Dummy topic for context sensitive Help on this menu command that jumps to WORKSHOP.HLP contents.

File | Open

This command displays the Open a File dialog box that lets you select a file to load into the Borland C++ IDE. Use this command to open a project (.PRJ or IDE), source file (.C or CPP), resource (.RC), script (.SPP or SPX), or any other type of file. The IDE automatically loads the file into the default viewer. For example, by default source files display in the Edit window and project files display in the Project window.

Opening multiple Edit windows

You can have multiple Edit windows open at the same time in the IDE, so the Open command does not close any of the Edit windows that you have open.

- If you load a file into an Edit window that contains an empty NONAME file, the IDE replaces the contents of the Edit window.

If you load a file that is already open in an Edit window, the Open command loads the file from disk in a new Edit window, leaving the original in the other Edit window. Subsequent changes are maintained in both copies.

File | Close

This command unloads the file in the active window. If the project window is active, this command unloads the current project and closes the project tree including all project files (nodes).

File | Save

This command saves the file in the active window. For example, if the Project window has focus, this command saves the current project (PRJ or IDE) file. If an Edit or Resource Workshop window has focus, this command saves the file in the active window.

Saving Files in the Edit window

If the file in the active Edit window has as a default name (such as NONAME00.CPP), the Borland C++ IDE opens the Save File As dialog box so you can rename the file as well as save it in a different directory or on a different drive.

If you use an existing file name to name the file, the C++ IDE asks if you want to overwrite the existing file.

- If you want to save all modified files, not just the file in the active window, choose File|Save All.

File | Save As

This command displays the Save File As dialog box, where you can save the file in the active Edit window under a different name, in a different directory, or on a different drive.

You can enter the new file name, including the drive and directory.

All windows containing this file are updated with the new name.

If you choose an existing file name, the Borland C++ IDE asks if you want to overwrite the existing file.

- This command is available only if an Edit or Resource Edit window is currently active.

File | Save All

This command works just like the Save command except that it saves the contents of all modified files loaded into an Edit window, not just the file in the active Edit window.

File | Print

See also

This command displays the Printer Options dialog box which lets you print the contents of the active Edit window.

Printer Options

Header/Page Number	Includes the name of the file, current date, and page number at the top of each page.
Line Numbers	Places line numbers in the left margin.
Syntax Print	Uses bold, italic, and underline characters to indicate elements with syntax highlighting.
Use Color	Prints colors that match colors on the screen (requires a color printer).
Wrap Lines	Uses multiple lines to print characters beyond the page width. If not selected, code lines are truncated and characters beyond the page width do not print.
Left Margin	Specifies the distance between the left edge of the page and the beginning of each line. For example, enter **2** to print with a two-character left margin **5** to print with a five-character left margin

The Print command is disabled if an Edit window is not currently active or if the IDE cannot detect a printer.

File | Printer Setup

This command displays the system Printer Setup dialog box where you select which printer you want to use for printing with the Borland C++ IDE.

This command is disabled if no printer can be detected.

File | Send

This command electronically mails a file from the active Edit window.

- This command is available only if you have a mail message service (MAPI) installed on your system.

File | Exit

This command exits the Borland C++ IDE and removes it from memory.

If you modified a source file without saving it, the IDE prompts you to do so before exiting.

List of closed files (File menu)

The last ten files closed during your editing session in the Borland C++ IDE appear at the bottom of the File menu so you can quickly reopen them.

Choose a file name at the bottom of the File menu to reopen it. When reopened, the file is removed from the list of closed files.

Edit menu

The Edit menu provides commands to undo, redo, cut, copy, paste, and clear text in the Edit and Resource Editor windows.

Undo command

Redo command

Cut command

Copy command

Paste command

Clear command

Select All

Buffer List

Edit | Undo

This command restores the file in the current window to the way it was before your most recent edit or cursor movement.

Undo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to a prior position.

If you undo a block operation, your file will appear as it was before you executed the block operation.

The Undo command will not change an option setting that affects more than one window or reverse any toggle setting that has a global effect; for example, Ins/Ovr.

- This command is available only if an Edit window is currently active and there is something to undo.

Edit | Redo

This command reverses the effect of the most recent Undo command.

Redo only has an effect immediately after an Undo command or another Redo command.

A series of Redo commands reverses the effects of a series of Undo commands.

- This command is available only if an Edit window is currently active and there is something to redo.

Edit | Cut

This command removes the selected text from your document and places the text in the Clipboard.

You can then choose Edit|Paste to paste the cut text into any other document (or somewhere else in the same document).

You can paste the cut text as many times as you want until you choose Edit|Cut or Edit|Copy.

- This command is available only if an Edit or Resource Workshop window is currently active and text has been marked for selection.

Edit | Copy

This command leaves the selected text intact and places an exact copy of it in the Clipboard.

To paste the copied text into any other document, choose Edit|Paste.

- This command is only available if an Edit window is currently active and text has been marked for selection.

Edit | Paste

This command inserts the contents of the Clipboard into the current window at the cursor position.

This command is available only if an Edit or Resource Editor window is currently active and there is something to paste.

Edit | Clear

This command removes the selected text but does not copy it to the Clipboard.

This means you cannot paste the text as you could if you had chosen Edit|Cut or Edit|Copy.

Although you cannot paste the cleared text, you can undo the Clear command with Edit|Undo.

Clear is useful if you want to delete text, but you do not want to overwrite text held in the Clipboard.

- This command is available only if an Edit window is currently active and text has been marked for selection.

Edit | Select All

This command selects the entire contents of the active Edit window.

You can then use Edit|Copy or Edit|Cut to copy it to the Clipboard, or perform any other editing action.

This command is available only if an Edit or Resource Editor window is currently active.

Edit | Buffer List

This command opens the Buffer List which lets you load a buffer into an Edit window.

Buffer List

The Buffer List displays a list of buffers. The buffer selected in the list is the file that had focus in the Edit window when Edit|Buffer List was selected.

If a file has been changed since it was last saved, the label **'M'** appears after the filename. If a file is read only, the label **'R'** is displayed.

Use the Buffer List to replace the contents of an Edit window without closing the original file. If the file you replace is not loaded in another Edit window, it is hidden. You can then later use the buffer list to load the hidden buffer into an Edit window.

Use the buttons on the Buffer List to perform the following actions:

Edit — Opens the selected buffer in the active Edit window, or

if an Edit window is not active and

- you select a buffer currently viewed in an Edit window, that window is brought to the foreground. If the window is minimized, it is restored.
- you select a buffer currently hidden, it is loaded into a new Edit window.
- This option is disabled when more than one buffer is selected.

Save — Writes the selected buffers to disk. Its status changes to unmodified and the label **(modified)** goes away.

Delete — Removes the selected buffers from memory, unless a buffer is loaded in an Edit window. If a buffer is loaded, the IDE prevents you from deleting it and the status line displays the message, **Buffer is viewed or in use**.

If the buffer is not loaded in an Edit window, but has been modified since it was last saved, the IDE prompts you to either save the file or delete it from the buffer list without saving.

Note: You can also use the Delete button to delete the selected buffers.

Close — Closes the Buffer List dialog.

Help — Displays this Help topic.

Buffer
A file loaded into memory which may or may not be currently loaded in an Edit window. Closing a window that contains the only view of a file also closes the buffer.

Search menu

The Search menu provides commands to search for text and error locations in your files.

Find
Replace
Search Again
Browse Symbol
Locate Symbol
Previous message
Next message

Search | Find

This command displays one of two dialog boxes depending on the which of the following windows is active:

- Edit window
- Message window

Search | Find (Edit window)

Lets you type the text you want to search for in the active Edit Window. Use the following options to affect the search:

Text to Find

Lets you enter the search string. Choose OK to begin the search, or choose Cancel to dismiss it. Press the Down arrow key or press Alt+Down arrow to display the history list associated with the input box.

Options

This group of options governs the kind of strings that the editor searches for.

Direction options

The Direction options specify which direction you want the editor to search, starting from the current cursor position. Forward (from the current position to the end of the file) is the default.

Scope

The Scope options determine how much of the file the editor searches in. Global (entire file) is the default scope.

Origin options

Origin specifies where the search should start. **From Cursor** is the default.

Text to Find

Use this area to enter a search string.

To begin the search, press ENTER or click OK. Click Cancel to dismiss it.

To enter a string that you've searched for previously, press Alt+Down arrow to show the history list, then choose from that list.

You can also pick up the word that your cursor is currently on in the Edit window and use it as the search string; simply invoke Find from the Search menu with your cursor on the word you want.

Find Text options

This group of options governs the kind of strings that the editor searches for.

- Case sensitive (Default)
- Whole words only
- Regular expression

Case Sensitive

When this option is on, the editor differentiates uppercase from lowercase when performing a search. Case Sensitive On is the default.

Whole Words Only

When this option is on, the editor searches for words only; the search string must have punctuation or space characters on both sides.

When the Whole Words Only option is off, the search might find the search string within longer words.

Regular Expression

When this option is on, the editor recognizes GREP-like wildcards in the search string.

To use Brief Regular Expressions instead:

1. Choose Options|Environment.
2. Select Editor Options.
3. Choose BRIEF Regular Expressions.

- The default is off.

Direction

Use these options to specify the direction you want the editor to search, starting from the Origin setting.

Forward	Searches toward the end of the file. (Default)
Backward	Searches toward the beginning of the file.

Scope

These options specify which part of the file you want to search in.

Global	Searches the entire file, in the direction specified by the Direction setting. (Default.)
Selected Text	Searches only in the block of marked text, in the direction specified by the Direction setting.

Use the mouse or block commands to mark a block of text. See Keyboard for more information.

Search | Find (Message window)

Enter the text you want to search for in the active Message window. If successful, the search takes you to the line containing the text and that line becomes highlighted.

To begin the search, press Enter or click OK. Click Cancel to dismiss it.

To enter a string that you've searched for previously, press Alt+Down arrow to show the history list, then choose from that list.

Use the following options to affect the search:

Start From

Case

Categories

Recurse Children

Start From

Choose...	To start searching from...
Top	the first entry in the Message window.
Current	the currently selected entry in the Message window.

The search continues to the last entry in the active Message window.

Case

Choose...	If you want the search to find occurrences ...
Sensitive	with the exact combination of uppercase and lowercase letters as the text you entered.
Insensitive	with the any combination of uppercase and lowercase letters as the text you entered.

Categories

Choose any combination of the following types of messages you want included in the search. For example, clear the Informational checkbox if you want to find specific text except when it is generated by an informational (status) message.

Category	Indicates
Informational	Progress such as build status.
Error	A problem that should be fixed, such as a missing declaration or a type mismatch.
Warning	A problem that can be overlooked.
Fatal	A problem of critical nature that prevents execution from continuing.

Recurse Children

Searches sublevel entries for all messages in the Message Window whether the view is expanded or not.

Origin options

These options determine where the search begins.

From Cursor

Starts the search at the cursor's current position. The search then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the Direction setting. (Default.)

Entire Scope

Searches either the entire block of selected text or the entire file (no matter where the cursor is in the file). The scope is defined by the Scope options.

Search | Replace

This command displays the Replace Text dialog box, where you type in the text you want to search for and the text you want to replace it with.

Dialog Box Options

The Replace Text dialog box is where you specify the text to search for and what to replace it with. Most components of the Replace Text dialog box are identical to those in the Find Text dialog box.

Text to Find

Enter the search string in the Text to Find input box, then click OK or Change All to begin the search. Click Cancel to stop it.

Press the Down-arrow key or press Alt+Down arrow to display the history list. associated with the input box.

New Text

Enter the replacement string in the New Text input box.

You can also use the history list to the right of the box to select a string you've used previously.

Options

This group of options governs the kind of strings that the editor searches for, and whether the replacement is automatic.

Direction

Use these options to specify which way you want the editor to search, starting from the current cursor position. Forward (from the current position to the end of the file) is the default.

Scope

Use these options to determine how much of the file the editor searches in. Global (entire file) is the default scope.

Origin

Use this option to specify where you want the search to start. **From Cursor** is the default.

Change All

Click Change All if you want the editor to replace all occurrences of the search string defined by the Direction, Scope, and Origin options.

New Text

After you enter the text you want to search for in Text to Find, use this area to enter the text you want to replace it with.

Press the Down-arrow key or press Alt+Down arrow to display the history list. associated with the input box.

To use the word your cursor is currently on in the Edit window as the search string; place the cursor on the word you want and then choose Replace from the Search menu .

Replace Text options

This group of options governs the kind of strings that the editor searches for, and whether the replacement is automatic.

Case sensitive

Whole words only

Regular expression

Prompt on Replace

Prompt on Replace

When this option is on, the editor asks you to confirm each replacement before it is made.

Note: When you are asked to confirm each replacement, the Change Allbutton is displayed on the message box. Press Change All to change all occurrences of the search string.

When this option is off, the editor replaces text matching the search string with the new text without confirmation.

When Prompt on Replace is off and you choose Change All, the editor replaces all occurrences of the search string.

- The default is on.

Change All

After you enter a search string and replacement string, you can choose OK or Change All to begin the search; or choose Cancel to stop it.

If you choose Change All and the Prompt on Replace option is:

on the editor asks you to confirm each replacement.

off the editor replaces all occurrences of the search string.

Search | Search Again

The Search Again command repeats the last Find or Replace command.

The last settings made in the Find Text or Replace Text dialog box remain in effect when you choose Search Again.

- In default keyboard mode, press F3 to search again.

Search | Browse Symbol

See Also

This command opens the Browse Symbol dialog box where you enter a symbol to display in the Browser.

When you choose this command from an open Edit window, the dialog box contains selected text, or if no text is selected, it contains the word at the cursor.

A symbol may be any class, function, or variable symbol which is defined in your source code (not only in the current file, but in any source file which is compiled and linked as part of the same project).

- This command is not available unless you have successfully compiled your program and included Debug and Browser information. See Using the Browser.
- This command works the same as the Browse Symbol command on the Edit window SpeedMenu.

Search | Locate Symbol

See Also

This command opens the Locate Symbol dialog box. Enter the symbol you want to locate and choose OK or press Enter.

The IDE positions the cursor in an Edit window at the line in the source file that contains the symbol or function.

- If the file is not currently loaded, the IDE opens it in an Edit window. Use Source Tracking to tell the IDE to open a new Edit window or use the current (active) Edit window when it loads a new file.

This command is only available while you are using the integrated debugger and your program is not running. For example:

This command is available if

- you are stepping through your code.
- your program is stopped at a breakpoint.
- you first choose Debug|Run and then choose Debug|Pause Program.

This command is not available if

- your program is currently running.
- you have not successfully compiled your program with Debug and Browser information. See Using the Browser.

Search | Previous Message

This command places the cursor on the line in your source code that generated the previous error or warning listed in the Message window.

- If the file is not currently loaded, the IDE opens it in a new Edit window.

Search | Next Message

This command places the cursor on the line in your source code that generated the next error or warning listed in the Message window.

- If the file is not currently loaded, the IDE opens it in a new Edit window.

View menu

The commands on the View menu let you choose which windows display in the IDE:

ClassExpert
Project
Message
Classes
Globals
CPU
Process
Watch
Breakpoint
Call Stack

View | ClassExpert

This command displays the ClassExpert window where you can add and manage classes in an AppExpert application.

- This command is not available unless the current project is an AppExpert application that you generated by choosing the File|New|AppExpert command.

View | Project

See also

This command opens the project tree where you can view a graphical representation of your project with project elements displayed as nodes (usually associated with files) in an expandable/collapsible hierarchy diagram.

- This command is not available until you have a project loaded in the IDE. If the Project window is closed, you can reopen it again with this command.

View | Message

The View|Message command opens the Message window which displays informational (status), warnings, and error messages.

Use the Tabs on the bottom of the window to select the Message window page you want to view. Press Ctrl+<*shortcut-letter*> to quickly move to a new Message window page. When you run a tool from the Tool menu (like GREP), a tab on the Message window appears, which you can use to display the output of that tool.

In the Message window, you can

- scroll through the output messages
- navigate to the program line referenced in the message
- track messages (highlight one program line after another in the source code as you view messages in the Message window.)
- search for text with the Search|Find command

To edit the program line:

Select the message that references the program line you want to edit and press Enter. The cursor moves to that program line in the source code.

To track program lines:

Select the message that references the program line you want to track, then press Spacebar.

How the IDE tracks program lines depends on how you set the source tracking options in Environment|Preferences. If the file referenced is not in an Edit window, the IDE opens the file and displays it either in a new Edit window or in the current Edit window.

To display the file in a new Edit window, choose New Window as the Source Tracking option (the IDE default setting.) To replace the current file in the active Edit window, choose Current Window as the Source Tracking option.

Message Window

The Message window displays informational (status), warning, and error messages generated as you compile and run your program. To open the Message window, choose View|Message. The Message window is displayed at the bottom of the IDE and extends across the entire IDE.

The Message window contains a minimum of three tabs:

Tab	Displays messages generated when you...
Buildtime	compile and link your program.
Runtime	run and debug your program.
Script	run a script.

- Additional tabs display for selected tools such as GREP.

Automatic Error Tracking

As you track up and down through the messages in the Message window, the cursor in the Edit window moves to the corresponding line in the source file.

Using the Message Window

You can use the Message window the following ways:

- When a message refers to a file that is not currently loaded in an Edit window, select the error and then press Enter or the Spacebar to load the file.
- To view the source code location of a selected error in an Edit window while keeping the Message window active, press the Spacebar.
- To make the Edit window active and place the cursor on the line that caused the error, press Enter.
- When there is an "Unresolved External" error message in the Buildtime tab of the Message window, select the error and press Enter to display the source file that has the external reference. The source file is displayed in an Edit window.
- To close the Message window, choose Close from its Control menu.
- To clear the Buildtime tab, press Ctrl+A.
- You can also display transfer program output in the Message window.

Message window SpeedMenu

The Message window SpeedMenu provides the following commands:

Edit

View

Delete

Delete All

Precious Toggle

Help

Save as text

User-Defined Warnings

The User-defined warnings compiler option allows user-defined warnings to appear in the IDE's Message Window. This option is on by default.

- In addition to messages which you can introduce with the #pragma message compiler syntax, the User-Defined Warnings option allows warnings introduced in third party libraries to be displayed. Remember to contact the vendor of the header file that issued the warning should you require additional help on third party messages.

Edit (SpeedMenu)

This command brings up an active Edit window with the cursor positioned on the line in your source code that caused the highlighted warning or error. If the file is not currently loaded, the IDE opens it in a new Edit window.

Double-click on the message in the Message window to achieve the same results.

This SpeedMenu command has no equivalent command on the menu bar.

View (SpeedMenu)

This command brings up a non-active Edit window displaying your source code at the line that caused the warning or error. If the file is not currently loaded, the IDE opens it in a new Edit window.

Delete (SpeedMenu)

This command deletes the selected message from the Message window.

Delete All (SpeedMenu)

This command clears all messages from the Message window.

Precious Toggle (SpeedMenu)

Selecting a message in the Message window and choosing Precious Toggle disables the Delete and Delete All commands in the SpeedMenu so the message cannot be deleted. Choosing Precious Toggle again re-enables the Delete and Delete All commands.

Help (SpeedMenu)

Displays help on the message highlighted in the message window.

- To get help on a message, you can also press F1.

Save As Text (SpeedMenu)

This command opens the **Save Messages** dialog box that lets you save messages displayed in the Message window. It provides the following options:

Select...	If you want to...
All branches	save all messages posted to the message window including all sublevel messages.
Selected Branches	save only the messages (including any sublevel messages) that you have selected in the Message window.
Clipboard	paste saved messages into the Windows clipboard.
File	write saved messages to disk. If you specify a file name with no path, the file will created the current project directory, or the default directory (such as BC5\BIN) if no project is loaded.

- If you specify a file that already exits, its contents will be overwritten.

Expand Children	include sublevel messages. If you clear this checkbox, only top level messages are saved.

View | Classes

See also

This command opens the Browsing Objects window which displays all of the classes in your application, arranged in a horizontal tree that shows parent-child relationships. The window is automatically sized to display as much of your class hierarchy as possible.

You can highlight any class in the display by using the arrow keys, or by clicking directly on the class name. You can then go to the source code that defines the highlighted class, inspect the functions and data elements of the highlighted class, or print the source code.

If the program in the current Edit window or the first file in your project has not yet been compiled, you must first compile your program before invoking the Browser.

- This command is not available unless you have successfully compiled your program and included Debug and Browser information. See Using the Browser.

View | Globals

See also

This command opens the Browsing Globals window which lists every variable in your program, in alphabetical order.

You can highlight any variable in the display by using the arrow keys, or by clicking directly on the variable name. You can then go to the source code that defines the highlighted variable or inspect the declaration of the highlighted variable.

If the program in the current Edit window or the first file in your project has not yet been compiled, you must first compile your program before invoking the Browser.

- This command is not available unless you have successfully compiled your program and included Debug and Browser information. See Using the Browser.

View | Watch

See also

This command opens the Watches window, which is part of the integrated debugger. The Watches window lists the watch expressions you have set and the current value of the evaluated expressions. As you move through your program while debugging, the values of the watch expressions change as your program updates the data values contained in the expressions.

You can add a watch expression from both the Watches window SpeedMenu and the Edit window SpeedMenu.

View | CPU

This command opens the CPU window, which gives you a low-level view of the program you are debugging.

When you open the CPU window from the View menu, the Disassembly pane is positioned at the location of the current execution point.

- This command is available only when you are running the integrated debugger.

View | Process

See also

This command opens the **Process** window which displays the processes currently running and thread IDs of all currently active processes.

The Process window SpeedMenu

The commands on this SpeedMenu change depending on the item you have selected in the Process window.

Command	Action
Make Processs\ Thread Current	Activates the selected process or thread.
Run Process	Executes the selected process.
Reset Process	Returns the program to the state it was in at the beginning of the current debugging session. For more information, see Debug|Reset this process.
Pause process	Halts execution of an application running in a debugging session.
Terminate Process	Stops the current debugging session. For more information, see Debug|Terminate this process.
CPU	Opens the CPU window and positions the execution point on the disassembled instruction that corresponds to the function definition in memory.
Watches	Opens the Watches window.
Call stack	Opens the Call Stack window.

View | Breakpoint

See Also

This command opens the Breakpoints window. This window lists all the currently set breakpoints.

View | Call Stack

See Also

This command opens the Call Stack window which shows the sequence of functions your program called to get to the function currently running.

Each entry in the Call Stack window displays the function name and the values of any parameters passed to it.

- This command is available only when you are running the integrated debugger.

Project menu

See also Add-on Products

The Project menu provides commands to create or modify a project, and to make, build, or compile your programs.

Open Project

Close Project

New Target

Compile

Make All

Build All

Generate Makefile

Stop Background Task

Project | Open Project

See also

This command displays the Open a Project dialog box, where you select and load an existing project file. This command works the same as the File|Open command except that it initially displays only Borland C++ Project (.IDE) file types.

You can load and use projects from previous versions of Borland C++ (.PRJ files for example). If you load an old Borland C++ project, the IDE converts the project to the new project format.

Project | Close Project

See also

This command unloads your current project including all project files (nodes) and closes the Project Tree window, if it is open.

Project | New Target

See also

This command displays the Add Target dialog box where you add a new target node and specify target information for it.

The node you create is added to the current project and placed at the bottom of the project tree. This new node is created as a stand alone target. You can move it or make it a child of another node in the project tree by using the Alt+UpArrow/Alt+DownArrow, or Alt+LeftArrow/Alt+RightArrow keys.

The options on the Add Target dialog box are:

Name

Enter the name for the new target node.

Type

Select from the following Target types:

AppExpert	Automatically generated code for an OWL-based application.
Standard	An Application (.exe) project with the current default project skeleton for that target type. If the target type is Standard, the Add Target dialog box appears where you further define your target.
Source Pool	A collection of nodes in the project that are typically marked "exclude from parent" and not built. Source Pools act as a model or template from which you can create reference copies of code for use throughout your project. By using Source Pools, you can let different targets use common source code. If the target type is Source Pool, the target is added to the project and you can add nodes to it immediately.

Project | Compile

See also

Project|Compile translates the file or selected project node.

When you select an Edit window, the project tree translates the file (i.e., if you have a **.rc** file open, it'll do a resource compile on it, if it's a **.c** or **.cpp** file, then a C or C++ compile on it).

When the project tree is in view and you select a node, the project tree translates only the selected node (not the whole project). The type of translation that takes place depends on the node type you select.

- When the selected node is a **.cpp** node, a C++ Compile takes place. To accomplish the same results, use the C++ Compile command on the Project tree window SpeedMenu which is the same as the -c command-line compiler option (compile and assemble, but do not link).
- When the selected node is an **.exe** node, the linker is called. To accomplish the same results, use the Link command on the SpeedMenu.
- When the selected node is a **.lib** node, the librarian is invoked. To accomplish the same results, use Special|Create Library command on the SpeedMenu.

A Compile Status information box displays the compilation progress and results. Choose OK when compilation is complete. If any errors occurred, the Message window becomes active and displays and highlights the first error or warning.

Compile Status information box

The Compile Status information box displays the compilation status and results.

You can change the speed of the compile using the Priority boost slide-bar to set the level of system resources that will be shared with other processes during build time:

-2 shares the maximum amount of available resources (longest compile time).

2 commits the maximum amount of resources to the build process (shortest compile time).

The Compile Information box displays the following information:

- Current directory
- Current file
- Lines compiled
- Number of warnings issued
- Number of errors generated
- Available memory

The information here is for display only; you cannot modify any of these settings.

To abort the compilation, choose Cancel.

After reviewing the information, choose OK to close the window.

Project | Make All

See also

This command makes the targets of the current project.

If you choose Make All, the project tree MAKEs all targets. It checks file dates and times to see if they have been updated. If so, the project tree rebuilds those files, then moves up the project tree and checks the next nodes' file dates and times. The project manger checks all the nodes in a project and builds all of the out-of-date files.

The .EXE file name is fully spelled out in the project tree for target names. If no project is loaded, the .EXE name is derived from the name of the file in the Edit window.

The Make All command rebuilds only the files that are not current. If you choose Make All after a failed Build All command, Make All rebuilds from the point of failure.

Project | Build All

See Also

The Build all command rebuilds all the files in the current project, regardless of whether they are out of date.

This option is similar to Project|Make All, except that Build All rebuilds all the files in the project whether or not they are current.

Build All builds the project using the Default Project Options Style Sheet unless you have attached a different Style Sheet to a node or overridden the options locally.

For example, if you have a project with an .EXE target that is dependent on a .CPP file, and the .CPP is dependent on two .H files, the project tree checks the dependencies for the two .H files first, then it compiles the .CPP file to an .OBJ, and finally, the project uses the new .OBJ file to link the .EXE.

Build All:

1. Deletes the appropriate precompiled header (.CSM) file, if it exists.
2. Deletes any cached autodependency information in the project.
3. Does a rebuild of the node.

If you abort a Build command by pressing Esc or choosing Cancel, or if you get errors that stop the build, you must explicitly select the nodes to be rebuilt.

Project | Generate Makefile

See Also

This command generates a make file for the current project. It gathers information from the currently loaded project and produces a MAKE file named <projectfilename>.MAK. You cannot convert makefiles to project files.

The IDE displays the new makefile in an Edit window.

Project | Stop Background Task

See Also

This command stops background compiles, links, makes, and builds if the compile is asynchronous.

Stop Background Task is only available after the compile, link, make, or build has started. It is disabled when the task is completed. This command is useful when the Status Box option (in the Options|Environment|Process Controls dialog) is set to None.

Script menu

Add-on Products

The Script menu provides the following commands:

Run

Commands

Modules

Compile File

Run File

Script | Run

See also

This command opens the Script Run window at the bottom of the IDE desktop which you use to enter a script command.

See the following topics for more information:

Referencing script functions

Script execution process

Referencing script functions

See also

A reference to a script function is processed as follows:

1. All loaded modules are searched for a matching function name. The search starts with the module most recently loaded and if unsuccessful, continues to the next most recently loaded module.
2. If found, the function executes. (If the function exists in more than one loaded module, the function located in the most recently loaded module is executed and other instances are ignored.)
3. If the function is not found, the IDE checks an internal table constructed by calls to the ScriptEngine SymbolLoad method. This table contains a list of script files and the predefined symbols they contain. If the function is found in the table, its associated module is loaded into the IDE and the script runs.
4. If no matching function is found, the IDE searches the Script Path for a script file name that matches the function name.
5. If a matching script file is found, it is loaded into the IDE and the script runs.
6. If a matching script file is **not** found, the IDE displays a message in the Script tab of the Message dialog box indicating that the function was not found.

- After the module is loaded, a second search for a function may be successful whereas the first search was not. For example, say a script file is found in the symbol table and gets loaded as a result of a function reference. The first search does not find the function, so the function does not execute. After the module is loaded, however, a second search finds the function in memory and it executes.

Script execution process

See also

When you load a module into the IDE, script execution takes place in the following order:

1. **global commands** — Script commands not in a function block.
2. **_init() function** — If a module contains an *_init()* function, it runs automatically, immediately after the global commands.
3. **autocall function** — If a module contains a function with the same name as the file in which it resides, it will execute automatically, immediately after the global commands and the *_init()* function (if any). This type of function is called an autocall function.

The script execution process lets you implement functionality without making changes to the STARTUP script, the IDE command line, or the Borland C++ configuration files.

For example, say you assign the F4 key to a function named *foo()* contained in a script called FOO.SPP. The first time you press F4, the IDE loads FOO.SPP and runs the function *foo()* because it is an autocall function. The second time you press F4, *foo()* executes immediately because the module is loaded and, therefore, is found in the symbol table. In both cases, pressing F4 executes the desired script.

Another example is a script named HELLO.SPP that contains a function called *hello()* declared as follows:

```
hello()
  {
    print "Hello World";
  }
```

When you load the script HELLO.SPP for the first time, the message `Hello World` displays in the Script tab of the Message window and the *hello()* function stays in memory. If you subsequently choose Script|Run and type `hello()` in the Script Run window and press Enter, the script processor calls the function *hello*() which displays `Hello World` in the Message window.

Script | Commands

See also

This command opens the Script Commands dialog box which displays a list of the available script commands and variables, including classes, functions, and global objects. If an object is an instance of a class, its properties and methods are also displayed.

To run a script command:

1. Double-click a command from the list.
2. Enter the arguments (if any) next to the selected command.
3. Click Run.

- To close the dialog box without executing a script command, click Close.

See the following topics for more information:

Referencing script functions

Script execution process

Click this button to execute the selected script.

Script | Modules

See also

This command opens the Module Management dialog box that lists all loaded modules (.SPP or .SPX files) as well as all modules in your Script Path and provides the following options:

Load

Unload

View

Browse

Help

Close

Reset

- To reload a module that is already loaded, double-click it.
- The options are persistent. For example, if you set the option to show only loaded scripts, the next time you open this dialog, only loaded scripts will display.

Filters and Legend

To choose which modules display, use the Filters and Legend options.

See the following topics for more information:

Referencing Script Functions

Script Execution Process

Filters and legend

Lets you choose which modules display in the Module Management dialog box.

Click...	If you want to display scripts...
Loaded Modules	already loaded into the IDE.
User	you have created and are already loaded into the IDE.
System	supplied with your Borland C++ installation and are already loaded into the IDE.
Unloaded Modules	found in your Script Path that have **not** been loaded into the IDE.

Legend

The text font on the Filter buttons indicate the following information:

bold	Loaded modules and user created scripts
italics	System supplied modules
normal text	Unloaded modules (both system and user created)

Module

By convention, the source files for scripts have the extension .SPP. When you load a script for the first time, it is compiled into an interpreted tokenized format called pcode. By default, a tokenized file is created that has the same name and the extension .SPX in the same directory as the script (SPP) file.

Loads the script into the IDE. After a script module has been successfully loaded, the IDE looks for the presence of two functions:

- a function named _*init*(),
- a function with the same name as the script file in which it is located (called an **autocall** function).

If a script contains these functions, they execute automatically, immediately after any global script commands; _*init* first, then the **autocall** function. If a series of scripts are loaded at the same time (on startup or from the command line), first all the _*init*() functions are processed (left to right), then the **autocall** functions.

Removes a script and its functions from memory.

Loads the selected script into an Edit window. If the script is already loaded in an Edit window, the script is loaded into a new Edit window.

Restarts the script engine as if you just started the IDE. All modules loaded during the current IDE session are unloaded and all scripts specified in Startup Scripts are reloaded.

Script | Compile File

See also

This command compiles the script in the active Edit window. If successful, the script is loaded into the IDE and runs.

Script | Run File

See also

This command executes the script in the active Edit window. If the script contains a breakpoint statement, the IDE passes control to the script debugger which opens the Script Breakpoint tool.

See the following topics for more information:

Referencing script functions

Script execution process

Tool Menu

See also Add-on Products

The Tool menu lets you run programs, tools, and utilities without leaving the IDE. To run a program from the Tool Menu, choose the program you want from the list of available tools.

Tools list

Items listed under the Tool menu show preinstalled or user-installed standalone programs (like GREP, Turbo Debugger, CodedGuard, or an alternate editor) that you can use to build nodes in the project, or viewers that let you display or edit the contents of a particular project node (file). To install other tools, use the Options|Tools dialog box.

When you choose a program from the menu, control transfers to it. Some programs, such as GREP and Turbo Debugger, transfer control back to the IDE after the program is through running.

Program arguments are specified in the Options|Tools dialog box and are passed when the program is invoked. If a program requires arguments to be entered at runtime, you are prompted for transfer arguments.

Sharing Tools

There are two ways to share tools between projects.

Inheriting Tools

When you create or assign tools, those tools remain with the project for which they were created; they do not get added to the list of predefined tools. If, however, you want a new project to use one of your custom Tools, you can do so by letting a new project inherit tools from another project.

To pass user-defined tools (and Style Sheets) from an existing project to a new project:

1. Edit your BCW5.INI file to make sure it includes the following line that lets the IDE inherit tools from an open project:
   ```
   [project] inherit=1
   ```
2. Open a project that contains the tools you want to pass to a new project.
3. Choose Project|New Project to create a new project.

At creation, the new project inherits the option settings and user defined tools of the project that was open.

Editing the Project Description Language files

You can also share tools across projects by editing the Project Description Language files (.PDL) associated with your projects.

▪ Be careful if you choose to edit .PDL files. If a .PDL file is corrupted, the project tree will not be able to read it. You should make a backup copy of the .PDL file before you begin making changes.

A .PDL file is a text file that contains information about the Style Sheets and Tools used in your project. Style sheet and Tools information can be copied from one .PDL file to another, allowing you to quickly modify Style Sheet and tools for your projects.

The IDE generates .PDL files only if the [PROJECT] section of your BCW5.INI file contains the following lines:

```
SaveAsText=1
ReadAsText=1
```

To edit the .PDL file:

1. Open the .PDL file containing the Tool you want to share. You can open the .PDL file using any text editor.
2. Search for the Subsystem = Tool
3. Copy the text for the tool you wish to share. You can copy more than one Tool or the entire Tool subsection if you like.
4. Open the .PDL file that is going to use the copied Tool.
5. Find the Subsystem = <tool>, then paste the copied text into the Tool subsection or replace the Tool subsection if you copied it all.
6. Open the .PDL file that is going to use the copied Style Sheet.
7. Find the section for tools, then paste the copied text to the end of the existing tools list.
8. Save the .PDL file that received the copied Tool. When you open a project file, the project manager checks the date of the .PDL file with the same name as the .IDE file. If the .PDL file is newer than the .IDE file, the contents of the .PDL file is added to the .IDE file.

Program arguments dialog box

See also

Lets you specify options that you want to pass to a program selected from the Tools menu. For example, if you choose Grep from the Tools menu, use the Program arguments dialog box to supply options as you would enter them on the command-line.

Debug menu

The Debug menu provides the following commands to run your program and use the integrated debugger:

Run

Load

Attach

Run to

Pause process

Reset this process

Terminate this process

Source at Execution Point

Add breakpoint

Breakpoint Options

Add watch

Evaluate

Inspect

View Locals

- The integrated debugger supports 32-bit applications only. Use the stand-alone Turbo Debugger to debug 16-bit applications.

Debug | Load

See also

This command opens the **Load Program** dialog box which lets you load a program and optionally specify any command-line options to pass to the application you want to debug. If you enter program arguments into this dialog box, the debugger automatically pastes them into the Arguments field on the Debugger page of the Environment Options dialog box.

To load a program,

1. Enter a Program name, choose one from the History list, or click **Browse** to locate the program you want.
2. Enter the command-line options you want (if any) and click OK.

- The program runs until it reaches the location corresponding to the symbol or function specified in the Run to ... on startup setting under Options|Environment|Debugger|Debugger Behavior. If you do not specify anything in the Run to...on startup field, the IDE pauses the loaded process before it executes its startup code.

Debug | Attach

See also

Use the Attach command to begin a debugging session on a process that is already running. This is useful when you know approximately when the problem occurs during program execution, but you are not sure of the corresponding location in the program source code.

Attach opens the **Attach to Program** dialog box. Use this dialog box to:

- attach the integrated debugger to a currently running program
- close and remove a program from system memory

The Attach to Program dialog box offers a complete list of all the processes running on your system. The list shows the Process IDs (PIDs) followed by the Program name. You can attach to and debug many types of running processes; however, you cannot attach to the following processes:

- The IDE (BCW.EXE)
- A process that is already being debugged
- A process spawned by the process being debugged (use the Debug Child Process option to debug these processes)

In addition to the process listing, the dialog box has the following button commands:

Kill

Highlight the process you want to kill, and choose Kill. The debugger will immediately terminate the process. Use caution with this command; terminating the wrong process will freeze your system.

Attach

Highlight the process you want to debug, and choose Attach. When you attach a program, the debugger pauses the selected process and displays the current execution point in either the Edit window or the CPU window.

- The Edit window displays if the process you attach to is running in a section of code that contains debug information and source code can be found in the Source Path setting of the Debugger page of the Environment Options dialog box.

Refresh

Choose Refresh to reload the list of running processes.

Debug | Run

See also

This command executes your program.

For 32-bit applications only:

This command runs your program and loads it into the integrated debugger. If the source code has been modified since the last compilation, the IDE recompiles and links your program.

For 16-bit applications only:

This command runs your program, but you cannot control your program's execution using the integrated debugger (all other Debug menu commands are unavailable). For example, you cannot use the Debug| Terminate command to stop program execution of a 16-bit application. Once you begin running a 16-bit application from the IDE, you must terminate the program through normal means, such as by running the program to completion.

Debug | Run to...

This command opens the **Run to** dialog box, which lets you specify the line and source file where you want to begin debugging. You must specify a line in a source file that contains 32-bit debug information. The debugger will then run your program at full speed, and pause when it reaches the source line you specify.

If you place the insertion point on a line your source when you choose this command, the File and Line are filled in for you.

- You can bypass this dialog box if you position the cursor on the line in a source file where you want to resume debugging, then choose the **Run to current** command on the Edit window SpeedMenu.

For 16-bit applications only:

This command runs your program as if you chose the Debug|Run command, but you cannot control your program's execution from the IDE. For example, you cannot use the Debug|Terminate command to close a 16-bit program. Once you begin running a 16-bit application from the IDE, you must terminate the program through normal means, such as by running the program to completion.

Specifies the name of the source file that contains the code to which you want to run.

Specifies the line number in the specified source file to which you want to run.

Debug | Pause process

See also

This command halts execution of an application running in a debugging session.

This command is only available while you are using the integrated debugger and your program is running.

Debug | Reset This Process

See also

This command returns the program (32-bit only) to the state it was in prior to the current debugging session. For example, use this command to restore all variables, members, and expressions modified during the current debugging session to the their original values.

- This command is only available while you have a 32-bit process loaded in the integrated debugger.

Debug | Terminate process

See also

This command

- Stops the current debugging session.
- Releases memory your program has allocated and some of the memory used for debugging information.
- Closes any open files that your program was using.
- This command is only available while you have a process loaded in the integrated debugger. For example, this command is available if
- you first choose Debug|Run.
- you are stepping through your code.
- your program is stopped at a breakpoint.
- This command is not available
- for 16-bit applications.
- if debugging is not in progress.

Debug | Source at execution point

See also

This command positions the cursor at the execution point in an Edit window. If you closed the Edit window containing the execution point, the IDE opens an Edit window displaying the source code at the execution point. (If no source is available at the execution point, the program displays in the CPU view.)

- This command is only available when the integrated debugger is paused inside the program you are debugging.

For example:

This command is available if

- you are stepping over or stepping through your code.
- your program is stopped at a breakpoint.
- you first choose Debug|Run and then choose Debug|Pause Program.

This command is not available

- if debugging is not in progress.
- for 16-bit applications.

This command works the same as the Source at execution point command on the SpeedMenu on the Edit window.

Debug | Add Breakpoint

See also

This command opens the Add Breakpoint dialog box which lets you enter a breakpoint.

When you choose this command, the debugger creates a Source breakpoint on the line number at the insertion point in the file loaded in the Edit window. You can use the default settings, or you can modify the default settings to create a custom breakpoint.

- This command is not available for 16-bit applications.

Debug | Breakpoint Options

See also

This command displays the Breakpoint Conditions/Action Options dialog where you can set the properties that control the behavior of a single breakpoint or one or more breakpoints using a specified breakpoint option set.

- This command is not available for 16-bit applications.

Debug | Add Watch

See also

This command opens the Add Watch dialog box which lets you set a watch expression.

When you choose this command from an active Edit window, the dialog box contains selected text, or if no text is selected, it contains the word at the cursor.

After you add the watch, the Watches window displays.

- This command is not available for 16-bit applications.

Debug | Evaluate

See also

This command opens the Evaluator window. This window lets you:

- Evaluate a variable or expression.
- View the value of any variable or other data item.
- Alter the value of simple data items.
- Use the Evaluator as a calculator at any time.

This command works the same as the Evaluate command on the SpeedMenu in the Edit window.

- This command is not available for 16-bit applications.

Debug | Inspect

See also

This command opens the Inspect Expression dialog box that lets you enter a variable or expression you want to examine while debugging your program.

For example:

This command is available if

- you are stepping into or stepping over your code.
- your program is stopped at a breakpoint.
- you first choose Debug|Run and then choose Debug|Pause Program.

This command is only available when the integrated debugger is paused in a program you are debugging.

- The command is not available if you are running a 16-bit application.

Debug | View Locals

See also

View Locals creates an Inspector window that displays the local variables defined in the current scope. The Inspector window will be continually updated.

Debug|View Locals is the same as entering `$Locals` in the Inspect Expression dialog box. This command only functions when you are in debug mode.

Options menu

Provides commands for changing various default settings for both your project and your environment (integrated development environment), installing or modify tools, working with any Style Sheets that are available for your project, and specifying what gets saved when you exit the Borland C++ IDE.

Most of the commands in this menu lead to a dialog box. The Project and Environment commands both open a dialog box that lets you set options that affect your program and programming environment.

The commands on the Options menu are:

Project

Environment

Tools

Style Sheets

Save

Options | Project

Add-on Products

This command displays the Project Options dialog box, where you set options for your entire project.

The available topics are

Directories

Compiler

16-Bit Compiler

32-Bit Compiler

C++ Options

Optimizations

Messages

Linker

Librarian

Resources

Build Attributes

Make

Options | Environment

Add-on Products

Environment Options dialog box

This command displays the Environment Options dialog box, where you set options for the IDE in general.

The available topics are

Browser

Editor

Syntax Highlighting

SpeedBar

Scripting

Process Control

Preferences

Fonts

Project View

Debugger

Resource Editors

Setting options

The Style Sheet Options, Options|Project, and Local Options dialog boxes contain options that affect the active project. The Options|Environment dialog box contains options that affect your authoring environment.

How to set options

Options dialog boxes use a dual window format.

- Topics in the left window expand and collapse. (When you first open an Options dialog box, the list of topics is collapsed.)
- To set individual options, choose the sub-topic and options display on the right. You can change settings on one or several pages, without leaving the Options dialog box. Options are initially set to their default values.

Any settings you make take place immediately.

Click...	If you want to...
OK	accept changes to all options and close return to the Project Window.
Undo Page	restore the current screen of options to the settings present when you selected the Options Settings dialog box.
Cancel	restore all options to the settings present when you opened the Options Settings dialog box. (Same as close box.)
Help	get context-sensitive online Help. The Help button gives a summary for the current screen of options. For Help on an individual option, select (highlight) an option and press F1.

Options | Tools

See also Add-on Products

This command opens the **Tools** dialog box, where you can install, delete or modify the tools listed under the Tool menu. The Tool menu lets you run programming tools of your choice without leaving the Borland C++ IDE.

Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can run a DOS program with the Windows IDE transfer.

Tools

Displays all of the available tools.

New button

Defines a new tool and add it to the list of available tools. Pressing this button displays the Tool Options dialog box.

Copy button

Defines a new tool that has the same properties as the tool currently selected in the Tools list box. Pressing this button displays the Tool Options dialog box.

Edit button

Lets you change the properties of the tool currently selected in the Tools list box. Pressing this button displays the Tool Options dialog box.

Delete button

Removes the tool currently selected in the Tools list box from the list.

Translators, viewers, and tools

See also Add-on Products

Translators, viewers, and tools are internal and external programs that are available to you through the IDE.

- ***Translators*** are programs that create one file type from another. For example, the C++ compiler is a translator that creates .OBJ files from .CPP files; the linker is a translator that creates .EXE files from .OBJ, .LIB, .DEF, and .RES files.
- ***Viewers*** are programs that let you examine the contents of a selected node. For example, an editor is a viewer that lets you examine the source code of a .CPP file. Resource Workshop is a viewer for Windows resource files.
- ***Tools*** are programs that help you create and test your applications. Turbo Debugger and GREP are examples of programming tools.

The IDE associates each node in a project with different translators or viewers, depending on the file extension of the node. Although each node can be associated with several different translators or viewers, each node is associated with a single default translator or viewer. This is how the IDE knows to open the Edit window when you double-click a .CPP node (double-clicking a node invokes the default viewer on the node).

To see the default node type (determined by file extension) for a specific translator or viewer:

1. Choose Options|Tools to open the Tools dialog box.
2. Select the item you want to inspect from the Tools list.
3. Choose Edit to access the Tools Options dialog box.
4. Choose Advanced to access the Tool Advanced Options dialog box, then inspect the Default For text box.

When you right-click a node, you'll find that some source nodes have a Special command on the SpeedMenu. This command lists the alternative translators that are available for the node type selected. For example, the commands C To Assembler, C++ To Assembler, and Preprocess appear on the Special menu of a .CPP node. The command Implib appears if you selected a .DLL node.

Using the Special command, you can invoke any translator that is available for a selected node type. Also, by selecting a source node in the Project Tree and choosing Edit Node Attributes from the SpeedMenu, you can reassign the default translator for the selected node.

Tool Options dialog box

See also   Add-on Products

Lets you set the properties for a tool.

Name

Enter a short description of the tool you're adding. This is the name that appears on the Tool list. If a tool in the Name list box is highlighted when you select Edit, the IDE automatically fills in the other input boxes; otherwise, they are blank.

Path

Enter the program name. You can include the full path to the program, but the path is optional. Press the Browse button to search your drives and directories to locate the path and file name associated with the tool.

Command Line

This entry holds any command-line options, transfer macros, and IDE filters you want to pass to the program. IDE filters are .DLL files that let tools interface with the IDE (for example, the GrepFile tool uses a filter to output text to the Message window).

- Try using $PROMPT if you want to experiment with transfer macros. To see transfer macros and filters in use, choose Options|Tools, then select GrepFiles and choose Edit.

Menu Text

Enter the text you want to appear on the SpeedMenu and on the Tools main menu for the tool. If you want to assign a shortcut key to the item, precede the accelerator key character with an ampersand (&). For example, Compile &Help underlines the letter H, making it the shortcut key. If you want an ampersand in your menu text, use two (&&Up&date appears as &Update in the menu).

- You must supply Menu Text if you want the program item to appear on the SpeedMenu or Tools menu.

Help Hint

Enter the text you want to have appear in the help hint space at the bottom of the Tools dialog box. Help hint text appears in the status line when you select the menu item.

Advanced button

Displays the Advanced Tool dialog box.

Advanced Tool dialog box

See also

Use this dialog box to set the options for a tool. These options determine the menu on which the tool will appear and with which nodes in the project tree the tool is used.

Tool Type

Simple Transfer

Translator

Viewer

Tool Usage

Place on Tools Menu

Place on SpeedMenu

Place on File Open Menu

Place on File New Menu

Target Translator

Translator / Viewer Details

Translate From

Translate To

Default For

Simple Transfer

Makes the tool a simple transfer item. A simple transfer item is a general-purpose tool that can only be placed on the Tools menu.

Translator

See also

Makes the tool a translator that can be used for each file (or node) in a project.

A translator can be any program that changes (translates) one file type to another. For example, the compiler is a translator that uses .C and .CPP files to create .OBJs, and TLINK is a translator that uses .OBJ files to produce an .EXE file.

Viewer

See also

Makes the tool a viewer that will let you see the contents of a node selected in the project tree.

For example, an editor is a viewer that lets you examine the code in a .CPP file. On the SpeedMenu for a .CPP node, you'll see the Text Edit command. The default editor for the Text Edit view is the IDE editor.

Other node types have other viewers available. For example, Resource Workshop can view .RC files. You can't view an .EXE node in a text editor, but you can choose to view it using the integrated debugger, Turbo Debugger for Windows, the Browser, or even as an executing program.

Translate From / Applies To

Enter the file extensions for the types of nodes that can use the tool. defines the node types (determined by file extension) that a translator can translate. To specify multiple node types, use a semicolon to separate file extensions.

- When you enter a file extension in this field, the Project Manager adds the translator to the Special menu of the project nodes that have that file extension. When you choose Special from the Project Manager SpeedMenu, the Project Manager displays all the available translators for that node type. However, it's important that each node type can have only a single, default translator (see the description for Default For).
- To see how this works, look at the tool CppCompile (choose Options|Tools, double-click CppCompile, then click Advanced). The Tool Advanced Options dialog box shows that the C++ compiler is a translator for .CPP, .C, .CAS, and .H files. If you have a source node with a .C extension, CppCompile appears on the Special menu when you right-click the node and choose Special.

Translate To

Enter the file extension for the type of node that results from applying the tool to the node types listed in the Translate From input box.

Default For

Enter the file extensions for the types of nodes that use this tool as the default viewer/translator. You can enter more than one file extension by separating them with semicolons (;).

Place On Tool Menu

Places the tool on Tool menu.

- The text in the Menu Text input box on the Tool Options dialog box determines what appears on the menu.

Place On SpeedMenu

Places the tool on the project tree SpeedMenu.

- The text in the Menu Text input box on the Tool Options dialog box determines what appears on the menu.

Place On File Open Menu

Places the tool on File|Open menu.

- The text in the Menu Text input box on the Tool Options dialog box determines what appears on the menu.

Place On File New Menu

Places the tool on File|New menu.

- The text in the Menu Text input box on the Tool Options dialog box determines what appears on the menu.

Target Translator

If the Tool Type selected is:

Translator	Check Target translator if you want the translator to produce a final target (such as an .EXE file). When you use this translator, the node becomes a target and the translated file is saved to the Final directory (see Project | Options | Directories). If you don't check Target translator, the translated file is saved in the Intermediate directory.
View	Selecting the Target Translator option specifies that the tool can View only nodes that have been translated; the node must be translated before you can view it.

Adding Translators and Viewers

Use the Options|Tools commands to display the Tool Options dialog box. Use the fields in the Tool Options dialog box to add a tool.

- Although the Project Manager lets you define your own Tool items, these items apply only to the project that you add them to; they aren't added as permanent parts of the IDE. However, translators, viewers, and tools can be passed to new and existing projects by sharing the Style Sheets of the projects.

Options | Style sheets

This command displays the Style Sheets dialog box where you can specify default compile and run-time option settings associated with a project. Style Sheets are predefined sets of options that can be associated with a node.

Style Sheets dialog box

See also

You can create, compose, copy, edit, rename, or delete Style Sheets (predefined sets of options) in the Style Sheets dialog box.

Access this dialog box by choosing Options|Style Sheets on the main menu. You can also edit Style Sheets by clicking the Styles button on the Edit node attributes dialog box.

The following options are available:

Create

Enter a name for the new Style Sheet.

Compose

Select from available Style Sheets to create a Style Sheet that contains the combined options from one or more Style Sheets. You must first create a new Style Sheet, then click Compose. Select a Style Sheet you want included in your new Style Sheet, then click Add. Continue adding Style Sheets, then click OK when you're finished.

The Add button is available when a Style Sheet in the Available Style Sheets window is highlighted. First select a Style Sheet, then press the Add button. The Style Sheet and its option settings are added to the composite Style Sheet.

The Remove button is available when a Style Sheet in the composite Style Sheet is highlighted. First select the Style Sheet that you wish to remove, then press the Remove button. The Style Sheet and its option settings are removed from the composite Style Sheet.

- You cannot edit a composed Style Sheet, but you can click Compose again to add or delete Style Sheets from the Composed one.

Copy

Select a Style Sheet from the Available Style Sheets list, then press the copy button. Enter a name for the new Style Sheet, then click OK. You can now click Edit to change any of the copied options. Copying is a fast way to create a Style Sheet that closely resembles another--you only have to change the options you want.

Edit

Select Edit to display the Style Sheet Options dialog box where you can reset options for a particular Style Sheet.

Rename

Enter a new name for a selected Style Sheet, then click OK.

Delete

Delete removes the selected Style Sheet from the available Style Sheets list. **Warning!** Once a Style Sheet is deleted, it can not be restored.

Setting style sheet options

The Style Sheet Options dialog box displays when you select Options|Style Sheet which displays the Style Sheet dialog box and then you choose to edit an existing style sheet or create a new style sheet.

This dialog box contains the same set of options that the Options|Project dialog box, but offers one additional option - the Include box.

The Include box

The Include box allows you to include or exclude selected options from a style sheet. Click on an option to set it and the Include box is automatically checked. Click on an option to unset it and the Include box is automatically unchecked.

The Include box is grayed if no option is selected (highlighted). You can tab over to an option if you want to see if the Include box is checked, but do not want to change the setting or affect the Include box for that option.

The Include box is checked if an option is changed from the Borland default option settings (by you or TargetExpert). For Borland default settings, the Include box is not checked, but default settings are automatically inherited and will be part of a style sheet unless you use the Include box to explicitly exclude options from it. For example, if you want a Style Sheet to handle only where directories are located, you can go through the rest of the dialog box and click on all of the other preset options, then explicitly clear the Include checkbox.

The Include box can also help you recover if you become confused about what options are on and what options are off for a particular Style Sheet. Just edit the Style Sheet and explicitly set and Include or Exclude the desired options.

Setting options

For information on setting options, see Setting Options.

Options | Save

This command opens the Save Options dialog box that lets you specify which of the following IDE settings will be saved to a file (or files).

Environment
Click Environment to store all environment information to a file.

Desktop
Click Desktop to save the desktop files.

Project
Click Project to save your project files.

Messages
Click Messages to save error messages in the Message window.

Node-specific tools

See also

This selection from the project tree SpeedMenu is a node-specific tool. Node-specific tools are either translators or viewers.

Translators

Translators are programs that change (translate) one file type into another file type. For example, the compiler is a translator that uses .C and .CPP files to create .OBJs, and TLINK is a translator that uses .OBJ files to produce an .EXE file.

Viewers

Viewers let you see the contents of a selected node. For example, an editor is a viewer that lets you examine the code in a .CPP file. On the local menu for a .CPP node, you'll see the Text Edit command. The default editor for the Text Edit view is the IDE editor.

Other node types have other viewers available. For example, Resource Workshop can view .RC files. You can't view an .EXE node in a text editor, but you can choose to view it using the integrated debugger, Turbo Debugger for Windows, the Browser, or even as an executing program.

Adding Tools to the SpeedMenu

Use the Options|Tools commands to display the Tool Options dialog box. Use the fields in the Tool Options dialog box to add a tool.

Window menu

The Window menu contains window management commands.

Most of the windows you manage from this menu have all the standard window elements, such as scroll bars, Minimize and Maximize buttons, and a Control menu box.

To open windows, use the File|Open or File|New commands.

These are the Window menu commands:

Cascade

Tile Horizontal

Tile Vertical

Arrange Icons

Close All

Minimize All

Restore All

List of open windows

More Windows

Window | Cascade

This command stacks all open windows and overlaps them so that part of each underlying window is visible.

- The IDE desktop area in which windows are arranged is set by the script IDE.SetRegion method.

Window | Tile Horizontal

This command arranges your open windows from top to bottom without overlapping one another. If there are more than three open windows, the IDE arranges them in a manner that allows more width than height.

- The IDE desktop area in which windows are arranged is set by the script IDE.SetRegion method.

Window | Tile Vertical

This command arranges your open windows from left to right so that they display next to each other. If there are more than three open windows, the IDE arranges them in a manner that allows more height than width.

- The IDE desktop area in which windows are arranged is set by the script IDE.SetRegion method.

Window | Arrange Icons

This command rearranges any icons on the desktop. The rearranged icons are evenly spaced, beginning at the lower left corner of the desktop.

This command is useful when you resize your desktop that has minimized windows. It is unavailable when no windows are minimized.

Window | Close All

This menu provides commands to close some or all open windows.

Windows

Debugger Windows

Browser Windows

Edit windows

If you modified the contents of a window since it was last saved, the C++ IDE opens a dialog box that asks if you want to save before closing.

When prompted to save: and you choose...	**the IDE...**
Yes	prompts you to save the changed file
No	closes all Edit windows; changes are not saved
Cancel	leaves the window open with your changes

Window | Close All | Windows

This command closes all open windows.

Window | Close All | Debugger windows

This command closes all open debugger windows.

Window | Close All | Browser windows

This command closes all open Browser windows.

Window | Close All | Edit windows

This command closes all open Edit windows.

If you modified the contents of a window since it was last saved, the C++ IDE opens a dialog box that asks if you want to save before closing.

When prompted to save: and you choose...	the IDE...
No	closes all Edit windows; changes are not saved
Yes	prompts you to save the changed file
Cancel	leaves the window open with your changes

Window | Minimize All

This menu provides commands to minimize all or some open windows:

Windows
Debugger Windows
Browser Windows
Edit windows

Window | Minimize All | Windows

This command minimize all open windows.

Window | Minimize All | Debugger Windows

This command minimizes all open debugger windows.

Window | Minimize All | Browser Windows

This command minimizes all open Browser windows.

Window | Minimize All | Edit windows

This command minimizes all open Edit windows.

Window | Restore All

This menu provides commands to restore some or all minimized windows to their former size:

Windows
Debugger Windows
Browser Windows
Edit windows

Window | Restore All | Windows

This command restores all minimized windows to their former size.

Window | Restore All | Debugger Windows

This command restores all minimized debugger windows to their former size.

Window | Restore All | Browser Windows

This command restores all minimized Browser windows to their former size.

Window | Restore All | Edit windows

This command restores all minimized Edit windows to their former size.

Window | List of open windows

See also

The IDE displays a list of currently open windows at the bottom of the Window menu.

When you choose an open window, you make that window active.

Window | More windows

Use this command to list additional open windows in the IDE not listed at the bottom of the Window menu.

Help menu

The Help menu provides access to online Help, which displays in a special Help window. The Help system provides information on virtually all aspects of the Borland IDE (integrated development environment) and the Borland C++ language, libraries, and so on.

- One-line menu and dialog-box hints appear on the status line whenever you select a menu command or dialog box item.

For more information, see Using Help in Borland C++.

If you do not know how to use Help in Windows, choose Using Help from the Help menu.

The Help menu provides the following commands:

Contents
Keyword Search
Using Help
Test
Windows API
OWL API
About

Help | Contents

This command displays the Borland C++ online Help contents. It provides a comprehensive list that summarizes the organization of topics in the Help system.

For more information, see Using Help in Borland C++.

If you do not know how to use Help in Windows, choose Using Help from the Help menu.

Help | Keyword Search

Place your cursor on a word (a "token") in the active Edit window and press F1 (or choose Help| Keyword Search) to get either

- help about that token.
- a list of available topics when there are multiple topics to choose from.
- a dialog displaying the closest match if no related topics are found.

This command works the same as the Go to Help Topic command on the Edit window SpeedMenu.

You can get Help this way for most Borland C++ language elements such as

- reserved words
- global variables
- functions in the standard run-time, class, and ObjectWindows libraries
- Windows API

For more information, see Using Help in Borland C++.

If you do not know how to use Help in Windows, choose Using Help from the Help menu.

Help | Keyboard

Use this command to get Help about hot keys, keyboard commands, and keyboard shortcuts.

For more information, see Using Help in Borland C++.

- If you do not know how to use Help in Windows, press F1 from an active Help window.

Help | Using Help

This command displays information on how to use the Borland C++ Help system in Windows.

- If you do not know how to use Help in Windows, press F1 from an active Help window.

Using Help Command

Help | Windows API

Use this command to get information on using the Windows Application Program Interface. It displays the Win32 Developer's Reference Help.

Help | OWL API

Use this command to get information on using OWL (Object Windows Library). It displays the Object Windows Library Help.

Help | About

This command displays the About Borland C ++ window which shows copyright and version information.

About Borland C++

This window displays copyright and version information.

Click OK, press Esc, or press Enter to close it.

Press Enter or click OK to close the About box.

SpeedMenus

SpeedMenus offer quick access to currently available commands. To display a SpeedMenu, click the right mouse button (or press Alt+F10).

SpeedMenus are available in the following parts of the Borland C++ IDE:

Breakpoints window SpeedMenu

Browser SpeedMenu

CPU_Window

Call Stack SpeedMenu

Classes pane SpeedMenu (ClassExpert)

Data Inspector SpeedMenu

Edit pane SpeedMenu (ClassExpert)

Edit window SpeedMenu

Events pane SpeedMenu (ClassExpert)

Message window SpeedMenu

Options window SpeedMenu

Project window SpeedMenu

Watch window SpeedMenu

Breakpoints window SpeedMenu

The Breakpoints window SpeedMenu provides commands to change breakpoint properties, to inspect or edit source code at breakpoints, or to add, remove, disable, or enable breakpoints.

Edit Breakpoint

Add Breakpoint

View Source

Edit Source

Remove Breakpoint(s)

Disable Breakpoint(s)

- The list of available commands changes depending on what you have selected in the Breakpoints window.

View Source (SpeedMenu)

This command displays a non-active Edit window at the line in your source on which the selected breakpoint is set. If the file is not currently loaded, the IDE opens it in a new Edit window.

- This SpeedMenu command has no equivalent command on the menu bar.

Edit Source (SpeedMenu)

This command displays an active Edit window with the cursor positioned on the line in your source code highlighted breakpoint is set. If the file is not currently loaded, the IDE opens it in a new Edit window.

- Double-click on the breakpoint in the Breakpoints window to achieve the same results.
- This SpeedMenu command has no equivalent command on the menu bar.

Browser SpeedMenu

This SpeedMenu provides the following commands to use the Browser.

Edit Source
Browse Symbol
Browse References
Browse Class Hierarchy
Return to Previous View
Print Class Hierarchy
Toggle Window Mode

Edit Source (SpeedMenu)

This command displays an active Edit window with the cursor positioned on the line in your source code that references the highlighted symbol.

- If the file is not currently loaded, the IDE opens it in an Edit window. Use Source Tracking to tell the IDE to open a new Edit window or use the current (active) Edit window when it loads a new file.

Browse Symbol (SpeedMenu)

This command displays the declaration of the selected symbol in a Browser window.

Double-click on the symbol to achieve the same results.

The symbol may be any class, function, or variable symbol which is defined in your source code (not only in the current file, but also in any source file which is compiled and linked as part of the same project).

This command is disabled if the current target does not exist.

Browse References (SpeedMenu)

This command lists all references to the selected symbol.

Browse Class Hierarchy (SpeedMenu)

This command displays all the members of the selected class arranged in a horizontal tree structure to show parent-child relationships.

Return to Previous View (SpeedMenu)

This command takes the Browser back to the previous view.

This command is disabled when the Browser is in Multiple Window mode or if there is no previous view for the active Browser window to change to.

To change the Browser window mode, choose Toggle Window Mode on the Browser SpeedMenu.

Print Class Hierarchy (SpeedMenu)

This command prints the horizontal tree structure displayed in the Browsing Objects window that shows parent-child relationships of members of the selected class.

Toggle Window Mode (SpeedMenu)

Lets you switch between single or multiple Browser modes.

Mode	Description
Single Window	Lets you open only one Browser view at a time. The next time you select a Browser action, the Browser replaces the view you currently have open.
Multiple Window	Lets you open multiple Browser views at the same time. Each time you select a new Browser action, an additional Browser view opens.

Set options (SpeedMenu)

This command displays the Environment Options dialog box where you set options that affect the IDE in general.

Watches window SpeedMenu

The Watches window SpeedMenu provides commands to change watch properties, add or delete watches, and disable or enable watches.

Edit Watch

Add Watch

Remove Watch(es)

Disable Watch(es)

Project window SpeedMenu

Add-on Products

The SpeedMenus on the Project window provide most of the commands you need in the IDE.

Which SpeedMenu commands display depends on the node selected and the commands available for that node. For example, if you select a .CPP source file in the Project window, the SpeedMenu displays commands appropriate for that node type, such as C++ Compile, Build Node, View|Text Edit, and so forth. (Additional commands not listed here may also appear if you have installed add-on products.)

The Project window SpeedMenu provides the following commands:

View submenu	Edit Node Attributes	Edit Local Options
Add Node	Properties	View Options Hierarchy
Delete Node		

The following commands may also appear on the SpeedMenu, depending on the node type selected:

Make Node	C Compile	Special submenu
Preview Make	C++ Compile	TargetExpert
Build Node	Resource Compile	Properties
Link	Assemble	

View submenu (SpeedMenu)

Add-on Products

Displays the commands that are available to view the current project node.

Each node type can have different viewers. As you select different node types, different commands appear on the View submenu. Also, which commands appear depends on which viewers are available for the current project and are specified to appear on the Project window SpeedMenu.

To make the viewer available on the Project window SpeedMenu

1 Choose Options|Tools to open the Tools dialog box.

2 Click Edit.

3 Click Advanced.

4 Click Place on SpeedMenu.

Special submenu (SpeedMenu)

Displays the commands that are available to translate or register the current project node.

Each node type can have different translators. As you select different node types, different commands appear on the Special submenu such as:

- C to Assembler
- C++ to Assembler
- Preprocess
- Register OLE Server
- Unregister OLE Server
- Create Library
- ImpLib (Import Library)
- Rescan (AppExpert projects only)

Note: If you choose Preprocess, the resulting .I file is opened in an Edit window.

Specifying translators for a node type

The commands available for a node type depend on which translators are available for the current project and those you have specified to appear on the Project window SpeedMenu.

To specify which translators are available for a node type on the Project window SpeedMenu

1. Choose Options|Tools to open the Tools dialog box.
2. Click Edit.
3. Click Advanced.
4. Click Place on the SpeedMenu.

Add Node (SpeedMenu)

Adds a node to the current project in the Project window.

The Add Node command displays the Add to Project List dialog box. To add a node, use this dialog box to specify a file to add to the project and choose Open.

The type of node added depends on the extension of the file you specify. For example, if the file you choose has an extension of .CPP, the node will have all the default attributes of a .CPP node.

Delete Node (SpeedMenu)

Deletes the currently selected node in the Project window.

Make Node (SpeedMenu)

Makes the currently selected node in the Project window.

Build Node (SpeedMenu)

Builds the currently selected node in the Project window.

Preview Make (SpeedMenu)

Provides information about what files will be made if you choose the Make command for this node. Preview Make displays the out of date glyph, a clock with an 'x' on it, next to nodes that will be rebuilt.

The Preview Make command puts a Make Preview branch on the Buildtime page of the Message window. Expand the branch to see what the Make command will do if you choose it.

Link (SpeedMenu)

Links the currently selected node in the Project window.

The Link command appears on the Project window SpeedMenu when an .EXE, .DLL, or AppExpert project node is selected.

Assemble (SpeedMenu)

Assembles the currently selected project node in the Project window.

The Assemble command appears on the Project window SpeedMenu when an .ASM project node is selected.

C Compile (SpeedMenu)

Compiles the currently selected project node in the Project window to an .OBJ file.

The C Compile command appears on the Project window SpeedMenu when a C project node is selected.

- This command also appears on the SpeedMenu Special submenu when a .CPP, .CAS, or .H project node is selected.

C Compile\C++ Compile (SpeedMenu)

Compiles the currently selected project node in the Project window to an .OBJ file.

The C Compile command appears when a .C node is selected.

The C++ Compile command appears when a .CPP node is selected.

- These commands also appear on the Special submenu when a .CAS, or .H project node is selected.

Resource Compile (SpeedMenu)

Compiles the currently selected project node in the Project window to a .RES file.

The Resource Compile command appears on the Project window SpeedMenu when an .RC project node is selected.

Properties (SpeedMenu)

Displays the Project View page of the Environment Options dialog box.

Use the Project View page to determine the type of information you want appear next to each node in the Project window.

Register or Unregister OLE Server (SpeedMenu)

The Special|Register OLE Server and Special|Unregister OLE Server commands on the Project window SpeedMenu alter the operating system's OLE registration database.

Register OLE Server

Use Special|Register OLE Server to identify a selected file as an OLE server and add information about it to the registration database. Using Special|Register OLE Server makes it unnecessary to run an application simply for registration purposes and makes it possible to register a DLL.

Unregister OLE Server

Use Special|Unregister OLE Server to delete a selected file from the OLE registration database.

Outliner SpeedMenu

This SpeedMenu provides commands to expand and collapse the nodes and branches in the options outliner.

Command	Accelerator Key	Function
Expand/collapse node	Spacebar	Opens or closes the categories (branches) beneath a node (top level)
Expand branch	+	Opens levels beneath a branch
Collapse branch	-	Closes levels beneath a branch
Expand all	*	Displays all options (all levels beneath all nodes and branches)
Collapse all	/	Collapses all sublevels (all levels beneath all nodes and branches)

Outliner Symbols

+ indicates a collapsed node or branch (sublevel categories not displayed)

- indicates a fully expanded node or branch

Browsing through your code

See also

The Browser lets you search through your object hierarchies, classes, functions, variables, types, constants, and labels that your program uses. The Browser also lets you:

- Graphically view the hierarchies in your application, then select the object of your choice and view the functions and symbols it contains.
- List the variables your program uses, then select one and view its declaration, list all references to it in your program, or go to where it is declared in your source code.
- List all the classes your program uses, then select one and list all the symbols in its interface part. From this list, you can select a symbol and browse as you would with any other symbol in your program.

Using the Browser

If the program in the current Edit window or the first file in your project has not yet been compiled, the IDE must first compile your program before invoking the Browser.

If you try to browse a variable or class definition (or any symbol that does not have symbolic debug information), the IDE displays an error message.

- If you changed the following default settings on the Project Options dialog box, before you use the Browser, be sure to:

1. Choose Options|Project.
2. Choose Compiler|Debugging and check
   - Debug information in OBJs
   - Browser reference information in OBJs
3. Choose Linker|General and check Include Debug Information.
4. Compile your application.

Browser views

The Browser provides the following views:

Global Symbols

Objects (Class Overview)

Symbol Declaration

Class Inspection

References

Browsing symbols in your code

See also

You can browse any symbol in your code without viewing object hierarchies or lists of symbols first.

To do so, highlight or place the insertion point on the symbol in your code and choose Browse Symbol. from the Search menu or the Edit window SpeedMenu.

If the symbol you select is a structured type, the Browser shows you all the symbols in the scope of that type. You can then choose to inspect any of these further. For example, if you choose an object type, you will see all the symbols listed that are within the scope of the object.

Starting the Browser

See also

To start browsing through your code, choose one of the following menu or SpeedBar commands: From the main menu or the SpeedBar:

Search|Browse Symbol

View|Classes

View|Globals

Customizing the Browser

See also

Use the Environment Options dialog box to select the Browser options you want to use.

1. Choose Options|Environment.
2. Choose Browser.
3 Specify the types of symbols you want to have visible in the Browser using the Visible Symbols option.
4. Specify how many Browser views you can have open at one time. See single or multiple Browser window mode in the Browser Window Behavior option.

Browsing objects (class overview)

See also

Choose View|Classes to see an overall view of the object hierarchies in your application, as well as the small details.

The Browser draws your objects and shows their ancestor-descendant relationships in a horizontal tree. The red lines in the hierarchy help you see the immediate ancestor-descendant relationships of the currently selected object more clearly.

To see more detail about a particular object, double-click it. (If you are not using a mouse, select the object by using your arrow cursor keys and press Enter.) The Browser lists the symbols (the procedures, functions, variables, and so on) used in the object.

One or more letters appear to the left of each symbol in the object that describe what kind of symbol it is. See Browser filters and letter symbols.

Browser SpeedMenu

Once you select the global symbol you are interested in, you can use the following commands on the Browser SpeedMenu:

Edit Source

Browse Symbol

Browse References

Return to Previous View

Print Class Hierarchy

Toggle Window Mode

Browsing global symbols

See also

Choose View|Globals to open a window that lists every global symbol in your application in alphabetical order.

To see the declaration of a particular symbol listed in the Browser, use one of the following methods:

- Double-click the symbol
- Select the symbol and press Enter
- Select the symbol, choose Browse Symbol from the SpeedMenu

Search

The Search input box at the bottom of the window lets you quickly search through the list of global symbols by typing the first few letters of the symbol name. As you type, the highlight bar in the list box moves to a symbol that matches the typed characters.

Browser SpeedMenu

Once you select the global symbol you are interested in, you can use the following commands on the Browser SpeedMenu:

Edit Source

Browse Symbol

Browse References

Return to Previous View

Toggle Window Mode

Symbol declaration window

See also

This Browser window shows the declaration of the selected symbol.

Browser SpeedMenu

You can use the following commands on the Browser SpeedMenu:

Edit Source

Browse References

Browse Class Hierarchy

Return to Previous View

Toggle Window Mode

Class inspection window

See also

This Browser window shows the symbols (functions and variables) used in the selected class.

Browser SpeedMenu

Once you select the symbol you are interested in, you can use the following commands on the Browser SpeedMenu:

Edit Source

Browse Symbol

Browse References

Browse Class Hierarchy

Return to Previous View

Toggle Window Mode

Browser filters and letter symbols

See also

When you browse a particular symbol, the same letters that appear on the left that identify the symbol appear in a Filters matrix at the bottom of the Browser window. The Filters matrix has a column for each letter which can appear in the top or bottom row of the column.

Use the filters to select the type of symbols you want to see listed. (You can also use the Browser Options settings to specify the types of symbols you want to see listed.)

Letter	Symbol
F	Function
T	Type
V	Variable
C	Integral constants
?	Debuggable
I	Inherited from an ancestor
v	Virtual method

- In some cases, more than one letter appears next to a symbol. Additional letters appear to the right of the letter identifying the type of symbol and further describe the symbol:

To view all instances of a particular type of symbol,
Click the top cell of the column.

For example, to view all the variables in the currently selected object, click the top cell in the V column. All the variables used in the object appear.

To hide all instances of a particular type of symbol,
Click the bottom cell of the letter column.

For example, to view only the functions and procedures in an object, you need to hide all the variables. Click the bottom cell in the V column, and click the top cells in the F and P columns.

To change several filter settings at once,
Drag your mouse over the cells you want to select in the Filters matrix.

Browsing references

See also

This Browser window shows the references to the selected symbol.

Browser SpeedMenu

You can use the following commands on the Browser SpeedMenu:

Edit Source

Browse Class Hierarchy

Return to Previous View

Toggle Window Mode

Using the Borland C++ editor

See also

To open an Edit window, choose File|Open. You can open the same file in more than one window.

Once in the Edit window you enter text just as if you were using a typewriter. When you want to end a line, press Enter. To cut, copy, or paste blocks of text, you first need to select a block.

There are many ways to make an Edit window active:

- Click it.
- Press Alt+hyphen to display the active Edit window's Control menu.
- Press Crtl+F6 (default keyboard mapping) to cycle though the open windows.
- Choose the window from the list of open windows on the Window menu.

Edit window

See also

Edit windows are where you type in and edit your C or C++ code.

Parts of the Edit window

The Edit window consists of three areas:

- a margin on the left side of the Edit window displays symbols that indicate the state of each line in your source as you debug your program. See Editor symbols.
- the text area where you type. See Selecting text.
- the caption.
- When the edit file is modified, the caption displays '*' after the file name. When the file is saved, the '*' is removed.
- If the edit file is read-only, the caption displays an 'R' after the file name. Any changes you make will not be saved.

Note: When you close an edit file, the cursor position is saved - the next time you open that file, the cursor will revert to the last position.

Edit Window SpeedMenu

The Edit window SpeedMenu provides commands to open a file, search for a keyword in Help, go to a line number or a cursor location, single step through code, add or toggle a breakpoint, set a watch, inspect data elements, evaluate an expression, open the CPU view, or move the insertion point to the execution point.

Open source

Go to help topic

Go to line

Inspect

Watch

Run to Current

Statement Step over

Statement Step into

Toggle Breakpoint

Evaluate

Add Breakpoint

CPU View

Source at execution point

Browse Symbol

- Some SpeedMenu commands are available only at certain times. For example, the Inspect and CPU view commands appear only when you run or debug your program.

Open Source (SpeedMenu)

This command displays the Open a File dialog box so you can select a file to load into the Edit window. It works the same as the File|Open command on the menu bar.

You can have multiple files open at the same time in the editor, so the Open command does not close any of the Edit windows that you have open. (Opening a file, however, replaces an empty NONAME file in an active Edit window.)

If you load a file that is already open in an Edit window, the Open command loads the updated version in a new Edit window, leaving the original copy in other Edit window. Subsequent changes are maintained in both copies.

Go to Help topic (SpeedMenu)

Place your cursor on a word (a "token") in the active Edit window, then choose this command (or press F1) to get Help on that word.

This SpeedMenu command works the same as the Help|Keyword Search command on the menu bar.

You can get Help this way for most Borland's C++ language elements such as

- reserved words
- global variables
- functions in the standard run-time, class, and ObjectWindows libraries
- Windows API

Go to Line (SpeedMenu)

Use this command to position the cursor on a specific line in the source code in the active Edit window. It displays the Go To Line Number dialog box which asks to enter the line number you want to find.

Type the number of the line you want (or pick a previously entered line number from the history list), then press Enter (or choose OK).

- The current line number displays in the status line at the bottom of the IDE desktop.

This SpeedMenu command has no equivalent command on the menu bar.

Inspect (SpeedMenu)

This command opens the Inspector window and displays the data element at the cursor location in the active Edit window.

When you use the Inspect SpeedMenu command, the integrated debugger uses the location of the insertion point in the Edit window to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements which are not within the current scope of the execution point.

This command is only available while you are using the integrated debugger, such as stepping through your code.

- If the cursor is on a whitespace when you choose this SpeedMenu command, the IDE brings up the Inspect dialog box, from where you can enter an expression to inspect.

Watch (SpeedMenu)

This command sets a watch on the symbol at the insertion point, then opens the Watches window where you can view the watched symbol as you use the integrated debugger.

- If the cursor is on a whitespace when you choose this SpeedMenu command, the IDE brings up the Add Watch dialog box, from where you can create a new Watch.

Run to Current (SpeedMenu)

This command tells the debugger to execute your program normally (not step-by-step) until it reaches the line in your code on which the cursor is located.

1. In an Edit window, position the cursor on the line where you want to resume debugging control.
2. Choose Run to Cursor on the Edit window SpeedMenu.

You can use Run to Cursor as a way to restart your debugging session after you have already been stepping over or stepping into your code.

This SpeedMenu command works the same as the Debug|Run to command on the menu bar.

- The IDE issues an error if you try to run to a source location that is not an executable line of code. This can happen when the insertion pointer in the Edit window is on a comment, a blank line, or a declaration.

Statement step over (SpeedMenu)

See also

This command executes the next statement in your program which is the one highlighted by the execution point.

If the execution point is on a call to a function, the debugger runs that function at full speed, then places the execution point on the next instruction following the function call. Statement step over is useful when you know that a function being called works and you do not wish to execute it one instruction at a time.

Statement step into (SpeedMenu)

This command runs your program statement-by-statement, but if the debugger encounters a function call, it stops on the first line of the function instead of executing the function as a single step.

If the executing statement

- contains a call to a function accessible to the debugger, program execution pauses at the beginning of the function definition.
- does not contain a call to a function accessible to the debugger, program execution pauses at the next executable statement.

Use the Statement Step into command to move the execution point into a function called by the one you are currently debugging. Subsequent Statement Step into or Statement Step over commands will run the statements within the function. When the debugger leaves the function, it resumes evaluating the statement that contains the call.

The Statement Step into command only recognizes functions defined in a source file that has been compiled with the following options selected:

- Debug Information in OBJs in Compiler|Debugging
- Include Debug Information in Linker|General

Toggle Breakpoint (SpeedMenu)

This command inserts an unconditional breakpoint on the line at the insertion point in the active Edit window, or removes a breakpoint if one is set on the line at the insertion point.

- To change the conditions and actions of a breakpoint, see Creating conditional breakpoints.

Evaluate (SpeedMenu)

See also

The Evaluate command opens the Expression Evaluator dialog box that lets you:

- Evaluate a variable or expression.
- View the value of any variable or other data item.
- Alter the value of simple data items.
- Use the Expression Evaluator as a calculator at any time.
- This command works the same as the Debug|Evaluate command.

Add Breakpoint (SpeedMenu)

This command opens the Add Breakpoint dialog box which lets you enter a breakpoint. Use this command when you want to create a conditional breakpoint.

CPU View (SpeedMenu)

This command opens the CPU window with the Disassembly view displaying the instructions that correspond to the currently selected line in the Edit window. If the selected line does not contain executable code, the integrated debugger issues an error message.

If the selected statement in the Edit window gets disassembled into more than one set of instructions (as is the case with template functions), choosing this command causes the integrated debugger to open the CPU window with the cursor positioned on the first instance in memory that corresponds to the selected source.

Source at execution point (SpeedMenu)

This command positions the cursor at the execution point in an Edit window. If you closed the Edit window containing the execution point, the IDE opens an Edit window displaying the source code at the execution point.

- If no source is available at the execution point, the program displays in the CPU window.
- This command works the same as the Debug|Source at execution point command.

Browse Symbol (SpeedMenu)

See Also

This command opens the Browse Symbol dialog box with the symbol highlighted or at the insertion point in the active Edit window. If there is no text at the cursor, the dialog box displays with no symbol.

- This command works the same the Search|BrowseSymbol command.

Selecting text

See Also

You can select (highlight) text in the Edit window either with keyboard commands or a mouse.

From the keyboard:

- Press Shift while pressing any key that moves the cursor.

With a mouse:

- Drag the mouse pointer over the desired text. To continue the selection past a window edge, just drag off the side. The window will scroll.
- To select a single line, double-click anywhere in the line.
- To select line-by-line, click once and press the mouse button again and begin to drag.
- To extend or reduce the selecting, hold Shift and click.

Once you have selected text, the commands in the Edit menu become available.

- The maximum number of lines allowed in a file is 2,147,483,647. You can have approximately 6MB open across all files, subject to available swap space.
- The maximum line width in an Edit window is 1024 characters; the computer will beep if you try to type past that.

Editor symbols

See Also

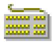
The margin (gutter) on the left side of the Edit window displays the following symbols (glyphs) that indicate the state of your program as you run it using the integrated debugger.

Symbol	Meaning
	You can step to or set a breakpoint on this line because it contains an executable statement
	Executable lines in your program shown while the process is running
	Line location of the execution point
	Unverified breakpoint
	Verified breakpoint
	Disabled breakpoint (verified and unverified)
	Invalid breakpoint
	Disabled invalid breakpoint
	Line where a process stopped running – you must first click Leave tracks in source where child stopped in the Options|Environment|Debugger|Debugger behavior settings

- Use the following settings to control the how the gutter displays:

  Visible gutter

  Gutter width



Keyboard

The Borland C++ keyboard shortcuts are one-, two- or three-key combinations you can press to perform a menu command or access a dialog box directly without using the mouse or the menus themselves. The function of specific keyboard shortcuts depends on which keystroke mapping scheme you have selected. Keystroke mapping schemes are contained in KBD files.

Click one of the following buttons to open a KBD file in Notepad:

- IDE default
- IDE classic
- Brief
- Epsilon

You can map your keyboard the following ways:

- Use the SpeedSettings on the Options|Environment|Editor page to have the Borland C++ IDE editor emulate one of four popular programming editors.
- Make your own settings by modifying or creating a KBD file.
- Use Script to assign a keystroke to a script command or function
- Changing the mapping of your keyboard may create conflicts with standard Windows keyboard commands. For example, the Brief keystroke mapping defines Alt+E as File|Open, while the standard Windows action for Alt+E is to activate the Edit menu. The mapped key takes precedence so that Alt+E allows you to open a file.
- In the IDE, the keystroke repeat rate defaults to OFF. To accelerate the keyboard repeat rate, add a the following information to your BCW5.INI file which is typically located in your Windows directory:

```
[Keyboard]
Accelerate=1
```

KBD file

See Also

Contains information used by KEYMAPS.SPP to map a key name to a script command or to assign a function to a key. KBD files can also contain option settings that affect groups of keys such as Persistent Blocks.

A KBD file has two sections: `[Options]` and `[Assignments]`.

The Options section can be any script command.

In the Assignments section, each line represents one assignment with each delimiter separated by `|:|` in the format:

```
KeyDescription |:| KeyCommand [|:| KeyFlag [|:| Comment]]
```

Where...	is...
KeyDescription	the key or key combination enclosed in angle brackets, for example: `<F1>` or `<Alt-Backspace>`
KeyCommand	the active view and function, for example: `editor.Undo();` indicates that in an active edit window, the key or key combination (specified in **KeyDescription**) is assigned to the Undo command
KeyFlag	one of the following values: ASSIGN_EXPLICIT (default) — No implicit assignments should be created. ASSIGN_IMPLICIT_KEYPAD — When an assignment is made to a sequence with a numeric keypad (Keypad) equivalent, such as PageUp, a second assignment is implicitly made for the equivalent. Assignments are made to both the shifted and non-shifted versions at the same time unless the implicit assignment would overwrite an explicit assignment. ASSIGN_IMPLICIT_SHIFT — <a> = = <A> ASSIGN_IMPLICIT_MODIFIER — <Ctrl+k><Ctrl+b> = = <Ctrl+k><b>
Comment	a description (optional)

Compiler command-line options

See also

The following table lists the command-line compiler options in alphabetical order:

Option	Description
@filename	Read compiler options from the response file "filename"
+filename	Use alternate configuration file "filename"
-1-	Generate 8086 compatible instructions
-1	Generate the 80186/286 compatible instructions (16-bit only)
-2	Generate 80286 protected-mode compatible instructions (16-bit compiler only) (Default for 16-bit)
-3	Generate 80386 protected-mode compatible instructions (Default for 32-bit)
-4	Generate 80386/80486 protected-mode compatible instructions
-5	Generate Pentium instructions
-A	Use only ANSI keywords
-a	Align byte (Default: -a- use byte-aligning)
-AK	Use only Kernighan and Ritchie keywords
-an	Align to "n" where 1=byte, 2=word (16-bit = 2 bytes), 4=Double word (32-bit only, 4 bytes), 8=Quad word (32-bit only, 8 bytes)
-AT	Use Borland C++ keywords (also -A-)
-AU	Use only UNIX V keywords
-B	Compile to .ASM (-S), the assemble to .OBJ (command-line compiler only)
-b	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
-C	Turn nested comments on (Default: -C- turn nested comments off)
-c	Compile to .OBJ, no link (command-line compiler only)
-Dname	Define "name" to the null string
-Dname=string	Define "name" to "string"
-d	Merge duplicate strings (Default)
-dc	Move string literals from data segment to code segment (16-bit compiler only)
-Efilename	Specify assembler
-efilename	Specify executable file name
-f	Emulate floating point
-f-	No floating point
-f87	Generate 8087 floating-point code (command-line compiler only)
-Fc	Generate COMDEFs (16-bit compiler only)
-Ff	Create far variables automatically
-Ff=size	Create far variables automatically; set the threshold to "size" (16-bit compiler only)
-ff	Fast floating point
-fp	Correct Pentium FDIV flaw

-Fs	Assume DS=SS in all memory models (16-bit compiler only)
-gn	Warnings: stop after "n" messages (Default: 255)
-H	Generate and use precompiled headers (Default)
-H=filename	Set the name of the file for precompiled headers
-H"xxx"	Stop precompiling after header file xxxx
-h	Uses fast huge pointers
-Hc	Cache precompiled header
-Hu	Use but do not generate precompiled headers
-in	Make significant identifier length to be "n" (Default)
-Jg	Generate definitions for all template instances and merge duplicates (Default)
-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances
-jn	Errors: stop after "n" messages (Default)
-K	Default character type unsigned (Default: -K- default character type signed)
-k	Turn on standard stack frame (Default)
-K2	Allow only two character types (signed and unsigned). Char is treated as signed. Compatibility with Borland C++ 3.1 and earlier.
-lx	Pass option "x" to linker (command-line compiler only)
-M	Create a Map file (command-line compiler only)
-mc	Compile using compact memory model (16-bit compiler only)
-mh	Compile using huge memory model
-ml	Compile using large memory model (16-bit compiler only)
-mm	Compile using medium memory model (16-bit compiler only)
-mm!	Compile using medium memory model; assume DS!=SS (16-bit compiler only. Note: there is no space between the -mm and the !)
-ms	Compile using small memory model (Default, 16-bit compiler only)
-ms!	Compile using small memory model; assume DS! = SS (16-bit compiler only. Note: there is no space between the -ms and the !)
-mt	Compile using tiny memory model (16-bit compiler only)
-N	Check for stack overflow
-O	Optimize jumps
-ofilename	Compile .OBJ to "filename" (command-line compiler only)
-O1	Generate smallest possible code
-O2	Generate fastest possible code
-Oa	Optimize assuming pointer expressions are not aliased on common subexpression evaluation
-Ob	Eliminate dead code
-Oc	Eliminate duplicate expressions within basic blocks
-Od	Disable all optimizations

-Oe	Allocate global registers and analyze variable live ranges
-Og	Eliminate duplicate expressions within functions
-Oi	Expand common intrinsic functions
-Ol	Compact loops
-Om	Move invariant code out of loops
-Op	Propagate copies
-OS	Pentium instruction scheduling
-Ov	Enable loop induction variable and strength reduction
-OW	Suppress the inc bp/dec bp on windows far functions (16-bit compiler only)
-P	Force C++ compile (command-line compiler only)
-p	Use Pascal calling convention
-pc	Use C calling convention (Default: -pc, -p-)
-po	Use fastthis calling convention for passing this parameter in registers
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention (32-bit compiler only)
-R	Include browser information in generated .OBJ files
-r	Use register variables (Default)
-rd	Allow only declared register variables to be kept in registers
-RT	Enable runtime type information (Default)
-S	Compile to assembler (command-line compiler only)
-Tx	Specify assembler option “x” (command-line compiler only)
-tD	Compile to a DOS .EXE file (same as -tDe) (command-line compiler only)
-tDc	Compile to a DOS .COM file (command-line compiler only)
-tW	Make the target a Windows .EXE with all functions exportable (Default)
-tWD	Make the target a Windows .DLL with all functions exportable
-tWDE	Make the target a Windows .DLL with explicit functions exportable
-tWE	Make the target a Windows .EXE with explicit functions exportable
-tWM	Generate a 32-bit multi-threaded target. This is the default for BCC32.EXE, the command-line compiler.
-tWS	Make the target a Windows .EXE that uses smart callbacks (16-bit compiler only)
-tWSE	Make the target a Windows .EXE that uses smart callbacks, with explicit functions exportable (16-bit compiler only)
-Uname	Undefine any previous definitions of "*name*" (command-line compiler only)
-u	Generate underscores (Default)
-V	Use smart C++ virtual tables (Default)
-v	Turn on source debugging
-V0	External C++ virtual tables
-V1	Public C++ virtual tables
-Va	Pass class arguments by reference to a temporary variable (16-bit compiler only)

-Vb	Make virtual base class pointer same size as **'this'** pointer of the class (Default, 16-bit compiler only)
-VC	Calling convention mangling compatibility
-Vc	Do not add the hidden members and code to classes with pointers to virtual base class members (16-bit compiler only)
-Vd	for loop variable scoping
-Ve	Zero-length empty base classes
-VF	MFC compatibility
-Vf	Far C++ virtual tables (16-bit compiler only)
-Vh	Treat "far" classes as "huge"
-vi	Control expansion of inline functions
-Vmd	Use the smallest representation for member pointers
-Vmm	Member pointers support multiple inheritance
-Vmp	Honor the declared precision for all member pointer types
-Vms	Member pointers support single inheritance
-Vmv	Member pointers have no restrictions (most general representation) (Default)
-Vo	Enable backward compatibility options (command-line compiler only)
-Vp	Pass the 'this' parameter to 'pascal' member functions as the first
-Vs	Local C++ virtual tables
-Vt	Place the virtual table pointer after nonstatic data members (16-bit compiler only)
-Vv	'deep' virtual basses
-WX	Compile to a DOS DPMI .EXE file (command-line compiler only)
-w	Display warnings on
-w"xxx"	Enable "xxx" warning message (Default)
-wamb	Ambiguous operators need parentheses
-wamp	Superfluous & with function
-wasm	Unknown assembler instruction
-waus	'identifier' is assigned a value that is never used (Default)
-wbbf	Bit fields must be signed or unsigned int
-wbei	Initializing 'identifier' with 'identifier' (Default)
-wbig	Hexadecimal value contains more than three digits (Default)
-wccc	Condition is always true OR Condition is always false (Default)
-wcln	Constant is long
-wcpt	Nonportable pointer comparison (Default)
-wdef	Possible use of 'identifier' before definition
-wdpu	Declare type 'type' prior to use in prototype (Default)
-wdup	Redefinition of 'macro' is not identical (Default)
-wdsz	Array size for 'delete' ignored (Default)
-weas	Assigning 'type' to 'enum'

-weff	Code has no effect (Default)
-wias	Array variable 'identifier' is near (Default)
-wext	'identifier' is declared as both external and static (Default)
-whch	Handler for '<type1>' Hidden by Previous Handler for '<type2>'
-whid	'function1' hides virtual function 'function2' (Default)
-wibc	Base class 'base1' is inaccessible because also in 'base2' (Default)
-will	Ill-formed pragma (Default)
-winl	Functions containing reserved words are not expanded inline (Default)
-wlin	Temporary used to initialize 'identifier' (Default)
-wlvc	Temporary used for parameter 'parameter' in call to 'function' (Default)
-wmsg	User-defined warnings
-wmpc	Conversion to type fails for members of virtual base class base (Default)
-wmpd	Maximum precision used for member pointer type <type> (Default)
-wnak	Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance)
-wnci	The constant member 'identifier' is not initialized (Default)
-wnfc	Non-constant function ‘ident’ called for const object
-wnod	No declaration for function 'function'
-wnst	Use qualified name to access nested type 'type' (Default)
-wntd	Use '> >' for nested templates instead of '>>' (Default)
-wnvf	Non-volatile function <function> called for volatile object (Default)
-wobi	Base initialization without a class name is now obsolete (Default)
-wobs	'ident' is obsolete
-wofp	Style of function definition is now obsolete (Default)
-wovl	Overload is now unnecessary and obsolete (Default)
-wpar	Parameter 'parameter' is never used (Default)
-wpch	Cannot create precompiled header: header (Default)
-wpia	Possibly incorrect assignment (Default)
-wpin	Initialization is only partially bracketed
-wpre	Overloaded prefix operator 'operator' used as a postfix operator
-wpro	Call to function with no prototype (Default)
-wrch	Unreachable code (Default)
-wret	Both return and return of a value used (Default)
-wrng	Constant out of range in comparison (Default)
-wrpt	Nonportable pointer conversion (Default)
-wrvl	Function should return a value (Default)
-wsig	Conversion may lose significant digits
-wstu	Undefined structure 'structure'

-wstv	Structure passed by value
-wsus	Suspicious pointer conversion (Default)
-wucp	Mixing pointers to different 'char' types
-wuse	'identifier' declared but never used
-wvoi	Void functions may not return a value (Default)
-wzdi	Division by zero (Default)
-X	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
-x	Enable exception handling (Default)
-xc	Enable compatible exception handling
-xd	Enable destructor cleanup (Default)
-xf	Enable fast exception prologs
-xp	Enable exception location information
-Y	Generate DOS overlay code (command-line compiler only)
-y	Line numbers on
-Yo	Overlay the compiled files (command-line compiler only)
-Z	Enable register load suppression optimization
-zAname	Code class set to "name"
-zBname	BSS class set to "name"
-zCname	Code segment class set to "name"
-zDname	BSS segment set to "name"
-zEname	Far segment set to "name"
-zFname	Far class set to "name"
-zGname	BSS group set to "name"
-zHname	Far group set to "name"
-zPname	Code group set to "name"
-zRname	Data segment set to "name"
-zSname	Data group set to "name"
-zTname	Data class set to "name"
-zVname	Far virtual segment set to "name" (16-bit compiler only)
-zWname	Far virtual class set to "name" (16-bit compiler only)

Command-line options by function

See also

The IDE groups the compiler and linker command-line options into the following categories:

Compiler

16-bit compiler

32-bit compiler

C++ options

Optimizations

Messages

Linker

In addition, there are compiler and linker options that you can set from only the command-line:

Command-line only options

Configuration Files

`@filename`	Read compiler options from the response file "filename"

Response Files

`+filename`	Use alternate configuration file "filename"

Compiler|Defines

`-Dname`	Define "name" to the null string
`-Dname=string`	Define "name" to "string"
`-Uname`	Undefine any previous definitions of "name"

Compiler|Code Generation

`-b`	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
`-K`	Default character type unsigned (Default: -K- default character type signed)
`-d`	Merge duplicate strings (Default)
`-po`	Use fastthis calling convention for passing this parameter in registers (16-bit compiler only)
`-r`	Use register variables (Default)
`-rd`	Allow only declared register variables to be kept in registers

Compiler|Floating Point

`-f-`	No floating point
`-f`	Emulate floating point
`-ff`	Fast floating point
`-fp`	Correct Pentium FDIV flaw

Compiler|Compiler Output

`-X`	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
`-u`	Generate underscores (Default)
`-Fc`	Generate COMDEFs (16-bit compiler only)

Compiler|Source

-C	Turn nested comments on (Default: -C- turn nested comments off)
-in	Make significant identifier length to be "n" (Default)
-AT	Use Borland C++ keywords (also -A-)
-A	Use only ANSI keywords
-AU	Use only UNIX V keywords
-AK	Use only Kernighan and Ritchie keywords
-VF	MFC compatibility

Compiler|Debugging

-k	Turn on standard stack frame (Default)
-N	Check for stack overflow
-vi	Control expansion of inline functions
-y	Line numbers on
-v	Turn on source debugging
-R	Include browser information in generated .OBJ files

Compiler|Precompiled Headers

-H	Generate and use precompiled headers (Default)
-Hu	Use but do not generate precompiled headers
-Hc	Cache precompiled header
-H=filename	Set the name of the file for precompiled headers
-H"xxx"	Stop precompiling after header file xxxx

16-bit Compiler|Processor

-1-	Generate 8086 compatible instructions
-1	Generate the 80186/286 compatible instructions (16-bit only)
-2	Generate 80286 protected-mode compatible instructions (16-bit compiler only) (Default for 16- bit)
-3	Generate 80386 protected-mode compatible instructions (Default for 32-bit)
-4	Generate 80386/80486 protected-mode compatible instructions
-a	Align byte (Default: -a- use byte-aligning)
-an	Align to "n" where 1=byte, 2=word (16-bit = 2 bytes), 4=Double word (32-bit only, 4 bytes), 8=Quad word (32-bit only, 8 bytes)

16-bit Compiler|Calling Convention

-pc	Use C calling convention (Default: -pc, -p-)
-p	Use Pascal calling convention
-pr	Use fastcall calling convention for passing parameters in registers

16-bit Compiler|Memory Model

-mt	Compile using tiny memory model (16-bit compiler only)
-ms	Compile using small memory model (Default, 16-bit compiler only)
-ms!	Compile using small memory model; assume DS! = SS (16-bit compiler only.

	Note: there is no space between the -ms and the !)
-mm	Compile using medium memory model (16-bit compiler only)
-mm!	Compile using medium memory model; assume DS!=SS (16-bit compiler only. Note: there is no space between the -mm and the !)
-mc	Compile using compact memory model (16-bit compiler only)
-ml	Compile using large memory model (16-bit compiler only)
-mh	Compile using huge memory model
-Fs	Assume DS=SS in all memory models (16-bit compiler only)
-dc	Move string literals from data segment to code segment (16-bit compiler only)
-Vf	Far C++ virtual tables (16-bit compiler only)
-h	Uses fast huge pointers
-Ff	Create far variables automatically
-Ff=size	Create far variables automatically; set the threshold to "size" (16-bit compiler only)

16-bit Compiler|Segment Names Data

-zRname	Data segment set to "name"
-zSname	Data group set to "name"
-zTname	Data class set to "name"
-zDname	BSS segment set to "name"
-zGname	BSS group set to "name"
-zBname	BSS class set to "name"

16-bit Compiler|Segment Names Far Data

-zEname	Far segment set to "name"
-zHname	Far group set to "name"
-zFname	Far class set to "name"
-zVname	Far virtual segment set to "name" (16-bit compiler only)
-zWname	Far virtual class set to "name" (16-bit compiler only)

16-bit Compiler|Segment Names Code

-zCname	Code segment class set to "name"
-zPname	Code group set to "name"
-zAname	Code class set to "name"

16-bit Compiler|Entry/Exit Code

-tW	Make the target a Windows .EXE with all functions exportable (Default)
-tWE	Make the target a Windows .EXE with explicit functions exportable
-tWS	Make the target a Windows .EXE that uses smart callbacks (16-bit compiler only)
-tWSE	Make the target a Windows .EXE that uses smart callbacks, with explicit functions exportable (16-bit compiler only)
-tWD	Make the target a Windows .DLL with all functions exportable
-tWDE	Make the target a Windows .DLL with explicit functions exportable

32-bit Compiler|Processor

-3	Generate 80386 instructions. (Default for 32-bit)
-4	Generate 80486 instructions
-5	Generate Pentium instructions

32-bit Compiler|Calling Convention

-pc	Use C calling convention (Default: -pc, -p-)
-p	Use Pascal calling convention
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention (32-bit compiler only)

C++ Options|Member Pointer

-Vmp	Honor the declared precision for all member pointer types
-Vmv	Member pointers have no restrictions (most general representation) (Default)
-Vmm	Member pointers support multiple inheritance
-Vms	Member pointers support single inheritance
-Vmd	Use the smallest representation for member pointers

C++ Options|C++ Compatibility

-Vd	for loop variable scoping
-K2	Allow only two character types (signed and unsigned). Char is treated as signed. Compatibility with Borland C++ 3.1 and earlier.
-VC	Calling convention mangling compatibility
-Vb	Make virtual base class pointer same size as 'this' pointer of the class (Default, 16-bit compiler only)
-Va	Pass class arguments by reference to a temporary variable (16-bit compiler only)
-Vc	Do not add the hidden members and code to classes with pointers to virtual base class members (16-bit compiler only)
-Vp	Pass the 'this' parameter to 'pascal' member functions as the first
-Vv	‘deep’ virtual basses
-Vt	Place the virtual table pointer after nonstatic data members (16-bit compiler only)
-Vh	Treat “far” classes as “huge”

C++ Options|Virtual Tables

-V	Use smart C++ virtual tables (Default)
-Vs	Local C++ virtual tables
-V0	External C++ virtual tables
-V1	Public C++ virtual tables

C++ Options|Templates

-Jg	Generate definitions for all template instances and merge duplicates (Default)
-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances

C++ Options|Exception Handling

-x	Enable exception handling (Default)
-xp	Enable exception location information
-xd	Enable destructor cleanup (Default)
-xf	Enable fast exception prologs
-xc	Enable compatible exception handling
-RT	Enable runtime type information (Default)

C++ Options|General

-Ve	Zero-length empty base classes

Optimizations

-Od	Disable all optimizations
-O1	Generate smallest possible code
-O2	Generate fastest possible code

Optimizations|16- and 32-bit

-Oc	Eliminate duplicate expressions within basic blocks
-Og	Eliminate duplicate expressions within functions
-Oi	Expand common intrinsic functions
-Ov	Enable loop induction variable and strength reduction

Optimizations|16-bit

-O	Optimize jumps
-Ol	Compact loops
-Z	Enable register load suppression optimization
-Ob	Eliminate dead code
-OW	Suppress the inc bp/dec bp on windows far functions (16-bit compiler only)
-Oe	Allocate global registers and analyze variable live ranges
-Oa	Optimize assuming pointer expressions are not aliased on common subexpression evaluation
-Om	Move invariant code out of loops
-Op	Propagate copies

Optimizations|32-bit

-OS	Pentium instruction scheduling

Messages

-w	Display warnings on
-wxxx	Enable "xxx" warning message (Default)
-gn	Warnings: stop after "n" messages (Default: 255)
-jn	Errors: stop after "n" messages (Default)

Messages|Portability

-wrpt	Nonportable pointer conversion (Default)

-wcpt Nonportable pointer comparison (Default)

-wrng Constant out of range in comparison (Default)

-wcln Constant is long

-wsig Conversion may lose significant digits

-wucp Mixing pointers to different 'char' types

Messages|ANSI Violations

-wvoi Void functions may not return a value (Default)

-wret Both return and return of a value used (Default)

-wsus Suspicious pointer conversion (Default)

-wstu Undefined structure 'structure'

-wdup Redefinition of 'macro' is not identical (Default)

-wbig Hexadecimal value contains more than three digits (Default)

-wbbf Bit fields must be signed or unsigned int

-wext 'identifier' is declared as both external and static (Default)

-wdpu Declare type 'type' prior to use in prototype (Default)

-wzdi Division by zero (Default)

-wbei Initializing 'identifier' with 'identifier' (Default)

-wpin Initialization is only partially bracketed

-wnak Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance)

Messages|Obsolete C++

-wobi Base initialization without a class name is now obsolete (Default)

-wofp Style of function definition is now obsolete (Default)

-wpre Overloaded prefix operator 'operator' used as a postfix operator

-wovl Overload is now unnecessary and obsolete (Default)

Messages|Potential C++ Errors

-wnci The constant member 'identifier' is not initialized (Default)

-weas Assigning ‘type’ to ‘enum’

-whid 'function1' hides virtual function 'function2' (Default)

-wnfc Non-constant function ‘ident’ called for const object

-wibc Base class 'base1' is inaccessible because also in 'base2' (Default)

-wdsz Array size for 'delete' ignored (Default)

-wnst Use qualified name to access nested type 'type' (Default)

-whch Handler for '<type1>' Hidden by Previous Handler for '<type2>'

-wmpc Conversion to type fails for members of virtual base class base (Default)

-wmpd Maximum precision used for member pointer type <type> (Default)

-wntd Use '> >' for nested templates instead of '>>' (Default)

-wnvf Non-volatile function <function> called for volatile object (Default)

Messages|Inefficient C++ Coding

-winl	Functions containing reserved words are not expanded inline (Default)
-wlin	Temporary used to initialize 'identifier' (Default)
-wlvc	Temporary used for parameter 'parameter' in call to 'function' (Default)

Messages|Potential Errors

-wpia	Possibly incorrect assignment (Default)
-wdef	Possible use of 'identifier' before definition
-wnod	No declaration for function 'function'
-wpro	Call to function with no prototype (Default)
-wrvl	Function should return a value (Default)
-wamb	Ambiguous operators need parentheses
-wccc	Condition is always true OR Condition is always false (Default)

Messages|Inefficient Coding

-waus	'identifier' is assigned a value that is never used (Default)
-wpar	Parameter 'parameter' is never used (Default)
-wuse	'identifier' declared but never used
-wstv	Structure passed by value
-wrch	Unreachable code (Default)
-weff	Code has no effect (Default)

Messages|General

-wasm	Unknown assembler instruction
-will	Ill-formed pragma (Default)
-wias	Array variable 'identifier' is near (Default)
-wamp	Superfluous & with function
-wobs	'ident' is obsolete
-wpch	Cannot create precompiled header: header (Default)
-wmsg	User-defined warnings

Linker options

General
Map file
16-bit linker
16-bit optimizations
32-bit linker
Warnings

Command-line only options

16- and 32-bit command-line options

16-bit command-line options

32-bit command-line options

Object search paths

See also

`(Command-line equivalent = /j)`

This option lets you specify the directories the linker will search if there is no explicit path given for an .OBJ module in the compile/link statement. This option works with both TLINK and TLINK32.

The Specify Object Search Path uses the following command-line syntax:

```
/j<PathSpec>[;<PathSpec>][...]
```

The linker uses the specified object search path(s) if there is no explicit path given for the .OBJ file and the linker cannot find the object file in the current directory. For example, the command

```
TLINK32 /jc:\myobjs;.\objs splash .\common\logo,,,utils logolib
```

directs the linker to first search the current directory for SPLASH.OBJ. If it is not found in he current directory, the linker then searches for the file in the C:\MYOBJS directory, and then in the .\OBJs directory. However, notice that the linker does not use the object search paths to find the file LOGO.OBJ because an explicit path was given for this file.

16- and 32-bit command-line options

See also

The following command-line switches are supported by the command-line compilers BCC.EXE and BCC32.EXE.

Compile to .ASM, then assemble

`(Command-line equivalent = -B)`

This command-line option causes the compiler to first generate an .ASM file from your C++ (or C) source code (same as the **-S** command-line option). The compiler then calls TASM (or the assembler specified with the **-E** option) to create an .OBJ file from the .ASM file. The .ASM file is then deleted. To use this 32-bit compiler option, you must install a 32-bit assembler, such as TASM32.EXE, and then specify this assembler with the **-E** option. In the IDE, right-click the source node in the Project Manager, then choose Special | C++ to Assembler.

- Your program will fail to compile with the **-B** option if your C or C++ source code declares static global variables that are keywords in assembly. This is because the compiler does not precede static global variables with an underscore (as it does other variables), and the assembly keywords will generate errors when the code is assembled.

Compile to .OBJ, no link

`(Command-line equivalent = -c)`

Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link on the resulting .OBJ files. In the IDE, choose Project | Compile.

Specify assembler

`(Command-line equivalent = -Efilename)`

Assemble instructions using *filename* as the assembler. The 16-bit compiler uses TASM as the default assembler. In the IDE, you can configure a different assembler using the Tool menu.

Specify executable file name

`(Command-line equivalent = -efilename)`

Link file using *filename* as the name of the executable file. If you do not specify an executable name with this option, the linker creates an executable file based on the name of the first source file or object file listed in the command.

Pass option to linker

`(Command-line equivalent = -lx)`

Use this command-line option to pass option(s) *x* to the linker from a compile command. Use the command-line option **-l-*x*** to disable a specific linker option.

Create a MAP file

`(Command-line equivalent = -M)`

Use this command-line option tells the linker to create a map file.

Compile .OBJ to *filename*

`(Command-line equivalent = -ofilename)`

Use this option to compile the specified source file to *filename*.OBJ.

C++ compile

`(Command-line equivalent = -P)`

The -P command-line option causes the compiler to compile all source files as C++ files, regardless of their extension. Use **-P-** to compile all .CPP files as C++ source files and all other files as C source files.

The command-line option **-P*ext*** causes the compiler to compile all source files as C++ files and it changes the default extension to whatever you specify with *ext*. This option is provided because some programmers use different extensions as their default extension for C++ code.

The option **-P-*ext*** compiles files based on their extension (.CPP compiles to C++, all other extensions compile to C) and sets the default extension (other than .CPP).

Compile to assembler

`(Command-line equivalent = -S)`

This option causes the compiler to generate an .ASM file from your C++ (or C) source code. The generated .ASM file includes the original C or C++ source lines as comments in the file.

Specify assembler option

`(Command-line equivalent = -Tx)`

Use this command-line option to pass the option(s) *x* to the assembler you specify with the **-E** option. To disable all previously enabled assembler options, use the **-T-** command-line option.

Undefine symbol

`(Command-line equivalent = -Uname)`

This command-line option undefines the previous definition of the identifier *name*.

16-bit command-line options

See also

The following switches are supported by the 16-bit command-line compiler (BCC.EXE) and linker (TLINK.EXE).

Generate 8087 instructions

(Command-line equivalent = **-f87**)

Use this 16-bit compiler option to create DOS 8087 floating-point code.

Compile to DOS .EXE

(Command-line equivalent = **-tD**)

The compiler creates a DOS .EXE file (same as **-tDe**).

Compile to DOS .COM

(Command-line equivalent = **-tDc**)

The compiler creates a DOS tiny-model .COM file. DOS .COM applications cannot exceed 64K in size, have segment-relative fixups, or define a stack segment. In addition, .COM files must have a starting address of 0:100H. If you change the file extension (to .BIN, for example), the starting address can be either 0:0 or 0:100H. The linker can't generate debugging information for .COM files, you will need to debug it as an .EXE file, then recompile and link the application as a .COM file.

Enable backward compatibility options

(Command-line equivalent = **-Vo**)

This compiler option enables the following 16-bit backward compatibility options: **-Va**, **-Vb**, **-Vc**, **-Vp**, **-Vt**, **-Vv**. Use this option as a handy shortcut when linking libraries built with older versions of Borland C++.

Compile to DPMI .EXE

(Command-line equivalent = **-WX**)

The compiler creates a DOS Protected Mode Interface (DPMI) .EXE file.

Generate overlay code

(Command-line equivalent = **-Y**)

The compiler generates the DOS code needed to create overlay files.

Overlay the compiled files

(Command-line equivalent = **-Yo**)

The compiler overlays the compiled overlay files that you specified with the **-Y** option.

Overlay the compiled files

(Command-line equivalent = **/o**)

This DOS only option causes TLINK to overlay all the modules or libraries that follow the option on the command line. Use **/o-** on the command line to turn of overlays. If you list a class name after this option, TLINK overlays all the segments in that class (this also works for multiple classes). If you don't list any name after this option, TLINK overlays all segments of classes ending with CODE. This option uses the default overlay interrupt number 3FH. To specify a different interrupt number, use **/oxx**, where *xx* is a two-digit hexadecimal number.

Specify option to RLINK

(Command-line equivalent = **/Rx**)

TLINK passes the option *x* to RLINK.EXE (*x* can be either **e**, **k**, **m**, **p**, or **v**).

Link DOS .COM

(Command-line equivalent = **/Tdc**)

TLINK generates a real-mode DOS .COM file. (The command-line option **/t** is supported for backward compatibility only; it has the same functionality as **/Tdc**.)

Link DOS .EXE

(Command-line equivalent = **/Tde**)

TLINK generates a real-mode DOS .EXE file.

Link Windows .DLL

(Command-line equivalent = **/Twd**)

TLINK generates a Windows .DLL file.

Link Windows .EXE

(Command-line equivalent = **/Twe**)

TLINK generates a Windows .EXE file.

Link DPMI .EXE

(Command-line equivalent = **/Txe**)

TLINK generates a DOS Protected Mode Interface (DPMI) .EXE file.

Expanded memory swapping

(Command-line equivalent = **/ye**)

This 16-bit linker option controls how TLINK uses expanded memory for I/O buffering. If the linker needs more memory for active data structures (while reading object files or writing the executable file), it either clears buffers or swaps them to expanded memory.

When reading files, the linker clears the input buffer so that its space can be used for other data structures. When creating an executable, it writes the buffer to its correct place in the executable file. In both cases, you can substantially increase the link speed by swapping to expanded memory. By default, swapping to expanded memory is enabled and swapping to extended memory is disabled. If swapping is enabled and no appropriate memory exists in which to swap, then swapping doesn't occur.

Default = ON

Extended memory swapping

(Command-line equivalent = **/yx**)

This TLINK option controls the linker's use of extended memory for I/O buffering. By default, the linker can use up to 8MB of extended memory. You can control the linker's use of extended memory with one of the following forms of this switch:

- **/yx** or **/yx+** uses all available extended memory, up to 8MB
- **/yx*n*** uses only up to ***n*** KB of extended memory

Default = OFF

32-bit command-line options

See also

The following switches are supported by the 32-bit command-line compilers (BCC32.EXE) and linker (TLINK32.EXE).

- The following 32-bit command-line options are not needed if you include a module definition file in your compile and link commands which specifies the type of 32-bit application you intend to build.

Generate a multi-threaded application

`(Command-line equivalent = -tWM)`

The compiler creates a multi-threaded .EXE or .DLL. (The command-line option **-WM** is supported for backward compatibility only; it has the same functionality as **-tWM**.). By default, this option is true for BCC32.

Link using 32-bit Windows API

`(Command-line equivalent = /aa)`

TLINK32 generates a protected-mode executable that runs using the 32-bit Windows API.

Link for 32-bit console application

`(Command-line equivalent = /ap)`

TLINK32 generates a protected-mode executable file that runs in console mode.

Link 32-bit .DLL file

`(Command-line equivalent = /Tpd)`

TLINK32 generates a 32-bit protected-mode Windows .DLL file.

Link 32-bit .EXE file

`(Command-line equivalent = /Tpe)`

TLINK32 generates a 32-bit protected-mode Windows .EXE file.

BCW command-line options

When you start BCW, you can specify options that direct the IDE's behavior. You can type these options on the command line or specify them as properties of the Borland C++ icon.

BCW has the following command-line syntax:

```
bcw [options] [filename]
```

`options` You can use either a hyphen (-) or a slash (/) to specify the options listed in the following table.

`filename` If you are not in the directory where `filename` is located, you must specify the full path to the file. BCW loads `filename` into the most logical environment based on the extension of the file. For example:

```
bcw cpp\project.ide
```

Displays the project in the project manager.

```
bcw resource\project.rc
```

Displays the resource file in the Resource editor.

```
bcw bc5\examples\windows\whello\whello.cpp
```

Displays the C++ source file in the Edit window.

```
bcw work\project.c
```

Displays the C source file in the Edit window.

You can specify the following BCW *options* in either upper- or lowercase:

Option	Description
-automate	If the IDE is set up as an OLE automation server, allows you to perform OLE-automated actions.
-b *project_filename*	Uses the IDE to build the specified *project_filename*. If the project consists of multiple targets, all targets are built. Once the targets are built, the IDE is closed. This option allows you to invoke the Borland C++ environment from a batch file so you can automate builds. For example: `bcw -b project.ide`
-d*process_ID*	Starts BCW and loads an already running process (specified by *process_ID*) into the integrated debugger. The IDE uses this command-line option to facilitate just-in-time debugging.
-i*ini_filename*	Starts BCW and specifies an INI file other than BCW5.INI. For example: `bcw -imyini.ini`
-m *project_filename*	Uses the IDE to make the specified *project_filename*. If the project consists of multiple targets, all targets are brought up to date. Once the targets are made, the IDE is closed. This option allows you to invoke the Borland C++ environment from a batch file so you can automate makes. For example: `bcw -m project.ide`
-q	Starts BCW quietly (with no splash screen or About box on startup).
-s *script_filename*	Starts BCW and starts the specified *script_filename*. For example: `bcw -s script.spp`

SpeedBar

The SpeedBar is a row of icons in the Borland C++ IDE that represents menu commands. Clicking one of the icons is a quicker alternative to choosing a command from the menu.

The icons that appear on the SpeedBar depend on your active window. Not all icons are available at all times.

The windows for which there are SpeedBars available include:

- Edit Window SpeedBar
- Browser SpeedBar
- Debugger SpeedBar
- Project Window SpeedBar
- Message Window SpeedBar
- IDE Desktop Speedbar
- ClassExpert Speedbar

The default placement of the SpeedBar is horizontally under the menu bar.

To change its appearance:

1. Choose Options|Environment.
2. Choose SpeedBar to select your SpeedBar Orientation.
3. Choose Customize to select your SpeedBar Options.

Debugging in the IDE

See also

No matter how careful you are when you code, your program is likely to have errors (bugs) that prevent it from running the way you intended. Debugging is the process of locating and fixing program errors that prevent your program from operating correctly. The Borland C++ IDE provides an integrated debugger that lets you debug your 32-bit Windows program without leaving the development environment.

- The integrated debugger does not support 16-bit Windows or DOS debugging; you must use TDW.EXE (Turbo Debugger for Windows) to debug 16-bit Windows programs and TD.EXE to debug DOS programs.

Types of errors

There are three basic types of program errors:

- Compile-time
- Run-time
- Logic

Using the integrated debugger

The integrated debugger can help you track down both run-time errors and logic errors. By running to specific program locations and viewing the state of your program at those places, you can monitor how your program behaves and find the areas where it is not behaving as you intended.

Although there are many ways to debug code, you will generally use a variation of the following process:

1. Run to a program location you would like to examine.
2. Examine the state of the program data values and view the program output.
3. Modify program data values to test bug fixes.
4. Reset the debugging session.
5. Fix the error.

- Before you begin a debugging session, compile your program with debugging information. Debugging information creates the connection between your program source code and the machine code produced by the compiler. Among other things, debugging information lets you view your source code in the Edit window while debugging.

Processes and threads

The integrated debugger supports debugging multi-threaded programs in both Windows NT and Windows 95. In addition, you can debug several processes at the same time. This functionality lets you debug client/server applications, where both the client and the server must be running at the same time.

- For the best integrated debugger performance, be sure to set the IDE Priority level of the compile to Normal. Because the debugger runs on a different thread than does the IDE, setting the IDE Priority to Normal ensures that both processes get equal status.

Running your program

All you need to do to begin a debugging session in the integrated debugger is to run your program from within the IDE. The integrated debugger gives you several ways to control the execution of your program.

When debugging, the main idea is to run your application to a point just before the location of an error. You can then examine the state of your program, and watch how it behaves as you move through the sections of code that are causing the problems.

With the integrated debugger, you can control the execution of your program in the following ways:

- Stepping through code
- Running to a breakpoint location
- Running to a program location
- Pausing your program

You can also use the integrated debugger to debug a process that you started running outside of the IDE. The IDE offers two different ways to begin a debugging session on an already running process:

- Attaching to a running process
- Using Just-in-time debugging to start the debugger when an application error occurs
- The integrated debugger does not support the debugging of 16-bit applications. If you run a 16-bit application from within the IDE (for example, if you choose Debug|Run), the application will run, but you will not be able to use the integrated debugger to control the execution of the application. Once you begin running a 16-bit application, you must terminate the program through normal means, such as running the program to completion.
- The IDE displays one of two glyphs in the status bar when you are debugging an application:
- If the application is running, a lightning bolt glyph is displayed.
- If you are stepping or hit a breakpoint, a pause process glyph is displayed.

Just-in-time Debugging

Both Windows NT and Windows 95 give the integrated debugger the ability to trap application exceptions when the IDE is not running. If your application encounters an operating system exception while running, the Windows Program Registry can automatically launch the integrated debugger inside the IDE. The integrated debugger then displays the source line where the exception occurred in the Edit window, and you can begin a debugging session from where the exception occurred.

- If the application that generates the exception does not contain a Borland symbol table (see Generating Debugging Information), the integrated debugger opens the CPU window with the execution point located on the instruction that generated the exception.

To set up just-in-time debugging, you must specify a debugger in the Windows program registry. Once set up, Windows starts the registered debugger in the event of an application error. This mechanism lets you connect the integrated debugger in the IDE to any process that fails.

When an unhandled exception occurs, Windows displays an Application Error dialog box. The dialog box provides an OK button, which you can choose to terminate the application. However, if you have registered a debugger in the program registry, the dialog box also contains a Cancel button. Choosing Cancel invokes the registered debugger.

To add the integrated debugger to the program registry:

1. Run the program JITIME.EXE (located in your Borland C++ BIN directory) from your 32-bit Windows operating system (Windows NT or Windows 95).
2. Click...

Click...	To...
BCW (IDE)	register the integrated debugger as the default debugger
TD32	register TD32.EXE as the default debugger
Dr. Watson	register DRWATSON.EXE (a Windows NT tool) as the default debugger
None	not register any debugger
Other	register the debugger of your choice

3. Click Confirm Invocation if you want the Application Error dialog box to display a Cancel button. Otherwise, Windows automatically starts the selected debugger when any application error occurs.
4. Click OK to register the selected debugger.

Examining program data values

After you have paused your application within the integrated debugger, you can examine the different symbols and data structures with regards to the location of the current execution point.

You can view the state of your program by:

- Watching program values
- Inspecting data elements.
- Evaluating expressions
- Viewing the low-level state of your program
- Viewing functions in the Call Stack window

You can also use the Browser to view the global variables and classes contained in your program.

Modifying program data values

Sometimes you will find that a programming error is caused by an incorrect data value. Using the integrated debugger, you can test a "fix" by modifying the data value while your program is running. You can modify program data values using:

- The Evaluator window.
- The Inspector window's Change SpeedMenu command
- A breakpoint Evaluate action, set from the Breakpoint Condition and Action dialog box
- The CPU window's Memory Dump pane

Debugging multi-threaded applications

The integrated debugger gives you two tools to help you debug multi-threaded and multi-process applications:

- The Processes window
- The Process/Thread control

While the integrated debugger supports multi-thread debugging, only a single thread can be “active” at a given time. The active thread is the one that responds to debugger commands such as stepping and expression evaluation. However, you can have several windows open, with each window displaying information related to a different program thread. For example, you can have two CPU windows open, with each showing the low-level state of different threads.

Both the Call Stack window and the CPU window are “thread aware,” meaning that they display information based on a particular thread.

The Processes window

To access the Processes window, choose View|Process. The Processes windows displays a hierarchy tree which lists all the processes and threads running in the integrated debugger. The currently active thread has an arrow next to the thread.

The Processes window lets you select which thread you want to make active. Double-click a thread to make that thread active; all stepping and expression evaluation will be done with regards to the execution point in that thread.

- You can also change the active thread by giving focus to a window that is displaying information on a different thread.

Tip: Whenever you are debugging multi-threaded programs, leave the Processes window open so you can always see which is the currently active thread.

The Process/Thread control

The Process/Thread control appears as a combo-box on the SpeedBar if there is more than one thread or process running in the integrated debugger. The Process/Thread control is similar to the Processes window in that it displays all the processes and threads currently running in the IDE.

You use the Process/Thread control to change which thread the IDE displays in one of the thread-aware windows. If a thread-aware window is active, then selecting a thread with the Process/Thread control changes the contents of that window to the thread you select. If another type of window has focus, then selecting a thread with this control gives focus to the last active thread-aware window, and updates its display to show the thread selected.

Resetting the debugging session

Sometimes while debugging, you will find it best to start the debugging session from the beginning of the program. The integrated debugger offers two ways to terminate the current debugging session, and start fresh:

- The Reset This Process command
- The Terminate Process command

Fixing program errors

Once you have found the location of the error in your program, you can type the correction directly into the Edit window; the change takes effect immediately. However, once you change a line of code in the Edit window, you must recompile and link the program to continue debugging. Because of this, you might want to wait until you finish the current debugging session before you make any fixes to your code.

Instead of fixing an error while debugging, you might want to test your fix by modifying data values using the debugger. This way, you don't have to recompile your program to see if your fix works.

Fixing syntax errors

If your code has compile-time (syntax) errors and you try to compile it, the Message window opens and displays the errors and warnings generated.

Correcting syntax errors

1. In the Message window, double-click on the error or warning that you want to fix.

   The IDE positions your cursor on the line in your source code that caused the problem.
2. Make your correction.
3. If your code has more than one problem, double-click on another error or warning in the
4. Choose Project|Make All to recompile your program.
5. Choose Debug|Run to verify that your program is operating correctly.

Choosing which warnings display in the Message window

See also

To choose which type of warnings you want to see in the Message window

1. Choose Options|Project and select Messages.

   The IDE displays all the various categories of messages and, on the right, settings that affect all messages.

2. Choose one of the following options:

- Check Selected and the IDE displays only the selected warning messages. (You will select the messages in Step 4.)
- Check All and the IDE displays all warning messages, not just the ones you selected.
- Check None and the IDE displays no warnings.

3. Limit the number of error and warning messages displayed in the Message window.

   In the Stop After boxes, specify the maximum number of error messages and warning messages you want displayed each time you compile. The number can be 0 to 255.

   If you enter 0, the IDE will not limit the number of messages or warnings it displays in the Message window.

4. Select a category under messages and then choose the available types of errors or warnings you want.

   For example, if you want the compiler to warn you about suspicious pointer conversions that might violate the ANSI standard, select ANSI Violations and then check the Suspicious Pointer Conversion option.

Generating debugging information

See also

To debug run-time errors and logic errors, you must first compile and link your program with debugging information (also known as a symbol table).

Adding debugging information to your .OBJ files

1. Choose Options|Project to open the Option Settings dialog box.
2. Expand Compiler and select Debugging.
3. Choose Debug Information in .OBJs. This is the default setting.

Adding debugging information in your .EXE files

1. Choose Options|Project to open the Option Settings dialog box.
2. Expand Linker and select General.
3. Choose Include Debug Information. This is the default setting.
4. To ensure that the debugger can accurately step into C++ source files, choose Options|Project|Compiler|Debugging and click Out-of-Line Inline Functions.

Turning debugging information off

When you have fully debugged your program, be sure to link the final executable files with debugging information turned off. This reduces the final size of your program files.

Debugging with program arguments

To pass run-time arguments to the program you want to debug

1. Choose Options|Environment|Debugger.
2. In the Run Arguments box, enter the arguments you want to pass to your program when it starts.

In addition to the Run Arguments box, you can also specify program arguments if you use the Debug|Load command to begin a debugging session.

Stepping (overview)

Stepping is the simplest way to move through your code one statement at a time. Stepping lets you run your program one line (or instruction) at a time – the next line of code (or instruction) will not execute until you tell the debugger to continue. After each step, you can examine the state of the program, view the program output, and modify program data values. Then, when you are ready, you can continue executing the next program statement.

There are two basic ways to step through your code:

Step Into	The Step Into command causes the debugger to walk through your code one statement at a time. If the execution point is located on a function call, the debugger moves to the first line of code that defines that function. From here, you can execute that function, one statement at a time. When you step past the return of the function, the debugger resumes stepping from the point where the function was called. Using the Step Into command to step through your program one statement at a time is known as single stepping.
Step Over	The Step Over command is the same as Step Into, except that if you issue the Step Over command when the execution point is on a function call, the debugger executes the function at full speed, and pauses the execution on the line of code following the function call.

Stepping granularity

The debugger steps over single lines of lines of code based on the following rules:

- If you string several statements together on one line, you cannot debug those statements individually; the debugger treats all statements as a single line of code.
- If you spread a single statement over multiple lines in your source file, the debugger executes all the lines as a single statement.
- To ensure that the debugger accurately represents your C++ source code while stepping, choose Options|Project|Compiler|Debugging and click Out-of-Line Inline Functions.

Stepping over code

See also Overview of Stepping

To Step Over code, choose Statement|Step Over from the Edit window SpeedMenu or press F8 (default keyboard mapping).

When you choose the Step Over command, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger executes that function at full speed, including any function calls within the function highlighted by the execution point. The execution point then moves to the next complete line of code.

Example

The following code fragment shows how Step Over works. Suppose these two functions are in a program that was compiled with debug information:

```
func_1() {
  statement_a;
  func_2();
  statement_b;
}

func_2() {
  statement_m;
  func_3();
}
```

If you choose Step Over when the execution point is on `statement_a` in `func_1`, the execution point moves to highlight the call to `func_2`. Choosing Step Over again runs `func_2` at full speed, moving the execution point to `statement_b`. Notice that when you step over `func_2`, `func_3` is also run at full speed.

As you debug, you can choose to Step Into some functions and Step Over others. Step Over is good to use when you have fully tested a function, and you do not need to single step through its code.

Stepping into code

See also   Overview of Stepping

To Step Into code, choose Statement|Step Into from the Edit window SpeedMenu or press F7 (default keyboard mapping).

When you choose Step Into, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger moves the execution point to the first line of code that defines the function being called.

If the executing statement calls a function that does not contain debug information, the debugger opens the CPU window and positions the execution point on the disassembled instruction that corresponds to the function definition in memory.

Example

The following code fragment shows how Step Into works. Suppose these two functions are in a program that was compiled with debug information:

```
func_1() {
  statement_a;
  func_2();
  statement_b;
}

func_2() {
  int customers;
  statement_m;
}
```

If you choose Step Into when the execution point is on `statement_a` in `func_1`, the execution point moves to highlight the call to `func_2`. Choosing Step Into again positions the execution point at the first line in the definition of `func_2`. Another Step Into command moves the execution point to `statement_m` , the first executable line of code in `func_2`.

When you step past a function return statement (in this case, the closing function brace), the debugger positions the execution point on the line following the original function call. Here, the debugger would highlight `statement_b` with the execution point.

As you debug, you can choose to Step Into some functions and Step Over others. Use Step Into when you need to fully test the function highlighted by the execution point.

Running to the cursor

You can tell the debugger you want to execute your program normally (not step-by-step) until a certain spot in your code is reached:

1. In an Edit window or CPU window, position the cursor on the line where you want to begin (or resume) debugging.
2. Choose Run To on the SpeedMenu of the Edit window or the Disassembly pane of the CPU window.

If you want to run to a location where a symbol is defined, use Locate Symbol to find the symbol, then run to that location.

Locating a symbol

See also

To quickly locate a function in your program

1. Choose Search|Locate Symbol.
2. Select an expression from the history list or type one in.
3. Click OK or press Enter.

The IDE positions the cursor on the line in the file where the symbol or function is defined.

- This command is available only after you have complied your program with debug information included.

Returning to the execution point

While you are debugging, you can

- browse through any file in any Edit window
- go to any place in your file
- open and close files

To quickly return to the location of the execution point from anywhere in the IDE, choose

Source at Execution Point from either the Debug menu or from the Edit window SpeedMenu of the Edit window, or use the button on the SpeedBar.

The debugger positions the cursor at the execution point. If you closed the Edit window containing the execution point, the IDE opens a new Edit window displaying the source code at the execution point.

Viewing program output

As you step through your program, you can view your program output in its application window. Arrange your windows so that you can see both your source code and your application window as you step.

- If the IDE desktop window and your application window overlap as you debug, you will see some flickering in your application window. If you arrange these windows so they do not overlap, program execution will be quicker and display will be smoother.

You can also use Alt+TAB to view your program. If your program is stopped, however, you can view it but you cannot not make changes to it. You can then use Alt+TAB again to switch back to the IDE.

Restarting your program

See also

Sometimes while debugging, you might need to start over from the beginning of your program. For example, if you have executed past the point where you believe there is a bug, it might be best to restart the debugging session.

To restart your program, choose Debug|Terminate process.. When you terminate the process, the IDE

- resets the integrated debugger so that running or stepping, begins at the start of the main program.
- does not change the location of the source code displayed in the Edit window so that you can easily position the cursor to run your program to the line you were on when you reset it.
- Do not interrupt the execution of your program if it is executing Windows kernel code.

Pausing a program

Instead of stepping through code, you can use a simpler technique to pause your program:

Choose Debug|Pause process and your program will stop executing.

You can then examine the value of variables and inspect data at this state of the program. When you are done, choose Debug|Run to continue the execution of your program.

General Protection Exception

If a general protection exception (GP exception) occurs while you run your program in the IDE, see General Protection Exception.

Watch expressions

See also

If you want to monitor the value of a variable or expression while you debug your code, add a watch to the Watches window. The Watches window displays the current value of the watch expression based on the scope of the execution point.

Each time your program's execution pauses, the debugger evaluates all the items listed in the Watches window and updates their displayed values.

The Watches window

To display the Watches window, choose View|Watch.

The Watches window lists the watches you are currently monitoring.

- Check the checkbox beside a watch to enable it.
- Clear the checkbox beside a watch to disable it.
- The Watches window will be blank if you have not added any watches.

The left side of the Watches window lists the expressions you enter as watches and their corresponding data types and values appear on the right. The values of compound data objects (such as arrays and structures) appear between braces ({ }).

- If the execution point steps out of the scope of a watch expression, the watch expression is undefined. When the execution point re-enters the scope of the expression, the Watches window again displays the current value of the expression.

Adding a watch

Sea Also

You can add a watch the following ways:

- Place the insertion point on a word in an Edit window and choose Watch from the Edit window SpeedMenu. The debugger adds a watch on the expression at the insertion point and opens the Watches window.
- Use the Add Watch dialog box to create a watch expression on any variable or expression available to the program you are debugging.

The Add Watch dialog box

The Add Watch dialog box lets you monitor the value of both simple variables (such as integers) and compound data objects (such as arrays). In addition, you can watch the values of calculated expressions that do not refer directly to memory locations. For example, you could watch the expression `x * y + 4`.

To create a watch expression using the Add watch dialog box

1. Choose Debug|Add watch or choose Add watch from the Watches window SpeedMenu.
2. Enter an expression into the Expression input box.
3. Click OK to add the watch or choose any of the following optional settings:

   Program

   Thread ID

   Advanced

- After you add the watch expression, the IDE automatically opens the Watches window if it is not already open.

Expression
Enter the name of the program variable or expression you want to watch.

Program

If you have multiple processes loaded and you want to restrict the watch expression to a specific program, enter the .EXE name of the program in which you are interested.

- If only one process is loaded, leave this option blank.

Thread ID

If you are debugging a multi-threaded program and you want to limit the watch expression to a specific thread, enter the name of the thread in which you are interested. This option is useful when a variable or expression is shared by more than one thread, but you are interested in watching only the instances that occur in a particular thread.

- If you are debugging a single-threaded program, leave this option blank.

Advanced
Click this button to open the Watch Properties dialog box if you want to change how a watch expression displays in the Watches window.

Changing watch properties

See also

To change the properties of a watch

1. Choose View|Watch, to open the Watches window.
2. Double-click a watch or choose Edit Watch form the Watches window SpeedMenu to open the Edit Watch dialog box.

Edit Watch dialog box

Use this dialog box to change the settings for a watch expression:

1. Either accept or change the information in either of the following options:

   Program

   Thread ID

2. Either

- Choose OK to save your changes and close the dialog box.
- Click Advanced to open the Watch Properties if you want to change how a watch expression displays in the Watches window.

Watch properties

See also

The following Watch options let you control how a watch displays in the Watches window:

Radix

Choose Radix to change the display of integers to either decimal or hexadecimal format.

Display as

Choose Display as to format the result of the watch expression.

Repeat Count

If you are watching an array, enter the number of array elements you want to watch.

If you set the Display As setting to Memory Dump, use the repeat count setting to specify the number of memory bytes you want to watch.

Significant

If you are watching a floating-point variable, enter the number of significant digits you want to display. You can specify a number from 2-18. The default is 7.

Radix

See also

Choose one of the following options to specify the display format of integers in a watch expression.

Option	Types affected	Description
Decimal	Integers	Shows integer values in decimal form, including those in data structures.
Hexadecimal	Integers	Shows integer values in hexadecimal with the 0x prefix, including those in data structures.

Display As

See also

By default, the debugger displays the result of a watch in the format that matches the data type of the expression..

For example, integer values normally display in decimal form. If you select the Hexadecimal button on the Watch Properties dialog box for an integer type expression, the IDE changes the display format from decimal to hexadecimal.

The following table describes the Watch Properties dialog box Display As options and their effects.

Option	Types affected	Description
Default	All	Shows the result in the display format that matches the data type of the expression.
Structure	Structures / Unions	Shows field names and unions as well as values such as X:1;Y:10;Z:5.
Character	Characters / Strings	Shows special display characters for ASCII 0 to 31. By default, such characters are shown using the appropriate C escape sequences (\n, \t, and so on).
Pointer	Pointers	Shows pointers in segment:offset notation with additional information about the address pointed to. It tells you the region of memory in which the segment is located and, if applicable, the name of the variable at the offset address.
String	Char, strings	Shows ASCII 0 to 31 as C escape sequences. Use only to modify memory dumps.
Floating point	Floating point	Shows the significant digits specified; from 2-18. The default is 7.
Memory dump	All	Shows the size in bytes starting at the address of the indicated expression. By default, each byte displays two hex digits. Use the memory dump with the character, decimal, hexadecimal, and string options to change the byte formatting. Use the Repeat Count setting to specify the number of bytes you want to display.

Repeat count

See also

If you are watching an array, enter the number of array elements you want to watch.

If you set the Display As setting to Memory Dump, use the repeat count setting to specify the number of memory bytes you want to watch.

Significant

See also

If you are watching a floating-point variable, enter the number of significant digits you want to display. You can specify a number from 2-18. The default is 7.

See Watching Data Elements

Watching data elements

See also

If you are watching a data element such as an array, you can display the values of the consecutive data elements.

For example, for an array of five integers named `xarray`, type the number 5 in the Repeat Count box of a Watch Properties window to see all five values of the array. An expression used with a repeat count must represent a single data element.

The integrated debugger views the data element as the first element of an array if it is not a pointer, or as a pointer to an array if it is a pointer.

Floating Point Display As option

If you select the Floating Point Display option, you can also indicate the number of significant digits you want displayed in the watch expression result. Specify the number of digits, from 2-18, in the Significant box. The default is 7.

Disabling and enabling watches

Evaluating many watch expressions can slow down the process of debugging. Disable a watch expression when you prefer not to view it in the Watches window, but want to save it for later use.

To enable or disable a watch

1. Choose View|Watch to open the Watch window.
2. Either

- Click the checkbox next to a watch to enable it.
- Clear the checkbox next to a watch to disable it.

To disable or enable selected watches

1. Hold down the Shift or Ctrl key and click on one or more watches in the Watch window.
2. Choose Enable or Disable watches from the Watch window SpeedMenu.

Deleting a watch

You can delete a watch the following ways:

1. Choose View|Watch to display the Watches window.
2. Select one or more watch expressions. (To make multiple selections, hold down the Shift or Ctrl key and click.)
3. Choose Remove Watch on the SpeedMenu.

Evaluating expressions

See also

You can use the Evaluator window to:

- evaluate an expression at any time
- modify the values of a variable at run time

To evaluate an expression,

1. Choose Debug|Evaluate.

   The Evaluator window displays.

2. Select an expression from the history list or type one in.
3. Choose Evaluate.

The Result box shows the current value of the evaluated expression.

You can evaluate any valid C or C++ expressions except those that contain

- symbols or macros defined with `#define`
- local or static variables not in the scope of the function being executed

Evaluating functions

You can call a function from the Evaluate window by evaluating the function in the Expression input box. Type the function name, parenthesis, and arguments just as you would type the function call into your program, but leave out the statement-ending semicolon. When you evaluate a function, the Result box displays the value which the function returns.

- Use caution when you evaluate functions while debugging an application; any side effects that occur as a result of the function evaluation will modify the data values of the program you are debugging. For example, if you evaluate a function that increments a variable, the new value of that variable will be reflected when you continue to step through your application.

Display Format

To change the display format of an expression, see Expression evaluator format specifiers.

Expression evaluator format specifiers

See also

By default, the debugger displays the result in the format that matches the data type of the expression. Integer values, for example, normally display in decimal form. To change the display format, type a comma (,) followed by a format specifier after the expression.

Example

Suppose the Expression box contains the integer value **z** and you want to display the result in hexadecimal:

1. In the Expression box, type `z,h`
2. Choose Evaluate.

Format Specifiers

The following table describes the Evaluator format specifiers.

Specifier	Types affected	Description
,C	Characters / Strings	**Character.** Shows special display characters for ASCII 0 to 31. By default, such characters are shown using the appropriate C escape sequences (\n, \t, and so on).
,S	Char / Strings	**String.** Shows ASCII 0 to 31 as C escape sequences. Use only to modify memory dumps.
,D	Integers	**Decimal.** Shows integer values in decimal form, including those in data structures.
,H or ,**X**	Integers	**Hexadecimal.** Shows integer values in hexadecimal with the 0x prefix, including those in data structures.
,Fn	Floating point	**Floating point.** Shows n significant digits where n can be from 2 -18. For example, to display the first four digits of a floating point value, type `,F4` If n is not specified, the default is 7.
,P	Pointers	**Pointer.** Shows pointers in segment:offset notation with additional information about the address pointed to. It tells you the region of memory in which the segment is located and, if applicable, the name of the variable at the offset address.
,R	Structures / Unions	**Structure/Union.** Shows field names and unions as well as values such as X:1;Y:10;Z:5.
,n**M**	All	**Memory dump.** Shows n bytes starting at the address of the indicated expression. For example, to display the first four bytes starting at the memory address, type `,4M.` If n is not specified, it defaults to the size in bytes of the type of the variable. By default, each byte displays as two hex digits. Use memory dump with the C, D, H, and S format specifiers to change the byte formatting.

Expression evaluator

See also

The Evaluator dialog box lets you

- Evaluate an expression using the symbols in your code.
- Modify a variable as you run your program.

Expression

Enter the expression you want to evaluate or pick one from the history list and then choose Evaluate.

Result

Displays the current value of the expression in the Expression box.

New Value

Lets you assign a new value to a variable in your application. Enter a value you want to assign to the data variable in the Expression box and then click Modify. The change takes effect immediately.

Canceling a modification

Click Close to cancel a modification any time before you select Modify. Although you cannot undo a value you have changed, you can restore a data item to its previous value. To do so, enter the previous value in the New Value input box and modify the expression again.

Expression

Enter the expression you want to evaluate or pick one from the history list and then choose Evaluate.

You can evaluate any valid C or C++ expressions except those that contain

- symbols or macros defined with `#define`
- local or static variables not in the scope of the function being executed

Display Format

To change the display format of an expression, see Expression evaluator format specifiers.

Result

Displays the current value of the expression in the Expression box.

- You must first enter an expression in the Expression box and choose Evaluate.

New value

Enter a value you want to assign to the expression in the Expression box and then choose Modify. The change takes effect immediately.

- You must first evaluate an expression.

Evaluate button

Enter an expression and then choose Evaluate to display its current value in the Result box.

Modify button

Enter a value in the New Value input box and then choose Modify to change the current value of the expression in the Expression input box. You must first evaluate an expression.

- The change takes effect immediately and you cannot undo or cancel the change after you choose Modify. To restore a value, however, you can enter the previous value in the New Value input box and modify the expression again.

Close
Choose Close to dismiss the Evaluator dialog box. You cannot undo a change to a variable after you choose Modify.

Modifying variables

While debugging, you can change the value of a variable in your program by using the Evaluator window.

To change the value of a variable,

1. Choose Debug|Evaluate.

   The Evaluator window displays.
2. Select an expression from the history list or type one in.
3. Choose Evaluate.

   The result box displays the current value of the expression.
4. Enter a value in the New Value input box.

   If you do not want to modify the variable, choose Close. You cannot undo a change to a variable after you choose Modify. To restore a value, however, you can enter the previous value in the New Value input box and modify the expression again.
5. To change the value of the variable, choose Modify.

   The new value takes effect immediately.
6. When you are finished, choose Close.

Usage

When you change the values of variables,

- You can change individual variables or elements of arrays or structures, but you cannot change entire arrays or structures with a single expression.
- The expression in the New Value box must evaluate to a result that is assignment-compatible with the variable you want to assign it to. A good rule of thumb is if the assignment would cause a compile-time or run-time error, it is not a legal modification value.
- You cannot directly modify untyped arguments passed into a function, but you can typecast them and then assign new values.

Warning: Modifying values, especially pointer values and array indexes, can have undesirable effects because you might overwrite other variables and data structures. These errors might not be immediately apparent.

Inspecting data elements

See also

You can examine the values in a data element the following ways:

To display an Inspector window directly from the Edit window

1. In the Edit window, position your cursor on the data element you want to inspect.
2. Choose Inspect from the SpeedMenu.

To inspect a data element from the menu bar

1. Choose Debug|Inspect to display the Inspect Expression dialog box.
2. Type the expression you want to inspect, then choose OK.

Scope

Unlike watch expressions, the scope of a data element in an Inspector window is fixed at the time you evaluate it:

- If you use the Inspect command on the Edit window SpeedMenu, the integrated debugger uses the location of the insertion point in the Edit window to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements which are not within the current scope of the execution point.
- If you use the Debug|Inspect command, the data element is evaluated within the scope of the execution point.

If the execution point is in the scope of the expression you are inspecting, the value appears in the Inspector window. If the execution point is outside the scope of the expression, the value is undefined.

Inspecting local variables

You can inspect all the local variables defined in the current scope by inspecting the expression `$Locals`.

Data types

The number of panes and the appearance of the data in the Inspector window depends on which of the following types of data you inspect:

- arrays
- classes
- constants
- functions
- objects
- pointers
- scalar variables (**int**, **float**, and so on)
- structures and unions

For example, if you inspect an array, you will see a line for each member of the array with the array index of the member. The value of the member follows in its display format, followed by the value in hexadecimal.

Isolating the view in an Inspector window

You can more closely inspect certain elements (such as classes, structures, and arrays) in the Inspector window to isolate the view to the member level:

1. Select an item in the Inspector window.
2. Choose Inspect on the Inspector window SpeedMenu.

The scope of the data element remains the same as it was when opened on the Inspector window.

To view the source code that references the element you are inspecting

Choose Go To from the Inspector window SpeedMenu (Go To is available only when you are inspecting a function).

Changing the value of Inspector items

To change a single data inspector item

1. Select an item in the Inspector window.
2. Choose Change on the Inspector window SpeedMenu.
3. Type in a new value, then choose OK.

Selecting a range of data items

If you are inspecting a data structure, it is possible the number of items displayed might be so great that you will have to scroll in the Inspector window to see data you want. For easier viewing, narrow the display to a range of data items.

1. Select an item in the Inspector window.
2. Choose Range on the Inspector window SpeedMenu.
3. Type in the new range of values and choose OK.

Inspect error

See also

The symbol you are trying to inspect is not recognized by the debugger. For example, it may be an invalid symbol or out of scope.

Inspector window

See also

Use this window to examine the value of a data element.

To display the Inspector window

1. In an Edit window, position the cursor on the data element you want to inspect.

2. Choose Inspect from the Edit window SpeedMenu.

3. Choose OK.

or

1. Choose Debug|Inspect to display the Inspect Expression dialog box.

2. Type in the expression you want to inspect.

3. Choose Inspect.

The number of panes and the appearance of the data in the Inspector window depends on the type of data you inspect. You can inspect the following types of data: arrays, classes, constants, functions, pointers, scalar variables (**int**, **float**, and so on), and structures and unions.

The Inspector window SpeedMenu

The Inspector window has the following SpeedMenu commands:

Inspect	Opens a new Inspector window on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.
Descend	This is the same as the Inspect SpeedMenu command, except the current Inspector window is replaced with the details which you are inspecting (a new Inspector window is not opened). To return to a higher level, choose History from the SpeedMenu.
Go to (file.line no.)	Gives focus to the Edit window with the insertion pointer positioned at the line in your source code that corresponds to the data item you are inspecting. If the source file is not currently loaded, the IDE open a new Edit window.

▪ This command is available only if the element you are inspecting is a function.

New Expression	Lets you inspect a new expression.
Range	Lets you specify how many data elements you want to view. This is useful when you are inspecting arrays.
History	Displays a history list of inspecting activity. Use this command to return to a previous Inspector view.
Change	Lets you assign a new value to a data item.
Show/Hide Methods	Toggles the display of the Methods pane that appears in the middle of the Inspector window and shows member functions if the item you are inspecting is a class or class object.
Show/Hide Type	Toggles the display of the Type pane that appears at the bottom of the Inspector window and shows type of data you are inspecting (such as **int**, **char**, or **long**).

Inspect expression

See also

Enter a data element you want to inspect. It must be a program variable or expression available to the debugger.

Scope

Unlike a watch expression, the scope of a data element is fixed at the time you open it in the Inspector window and never changes:

- If you use the SpeedMenu command, the data element is always evaluated within the scope of the line on which the data element appears.
- If you use the menu bar command, the data element is evaluated within the scope of the execution point.

If the execution point is in the scope of the expression you are inspecting, the value appears in the Data Inspector window. If the execution point is outside the scope of the expression, the value is undefined.

Inspecting local variables

You can inspect all the local variables defined in the current scope by inspecting the expression `$Locals`.

Data Types

The number of panes and the appearance of the data in the Inspector window depends on which of the following types of data you inspect:

- arrays
- classes
- constants
- functions
- objects
- pointers
- scalar variables (**int**, **float**, and so on)
- structures and unions

For example, if you inspect an array of integers, you will see a line for each member of the array with the array index of the member. The value of the member follows in its display format, followed by the hexadecimal value of the integer.

Set index range

See also

Lets you change the values of an array displayed in an Inspector window.

If you are inspecting a data structure, it is possible the number of items displayed might be so great that you will have to scroll in the Inspector window to see the data you are want. For easier viewing, narrow the display to a range of data items.

Starting Index
Enter a new value for the first element in the array.

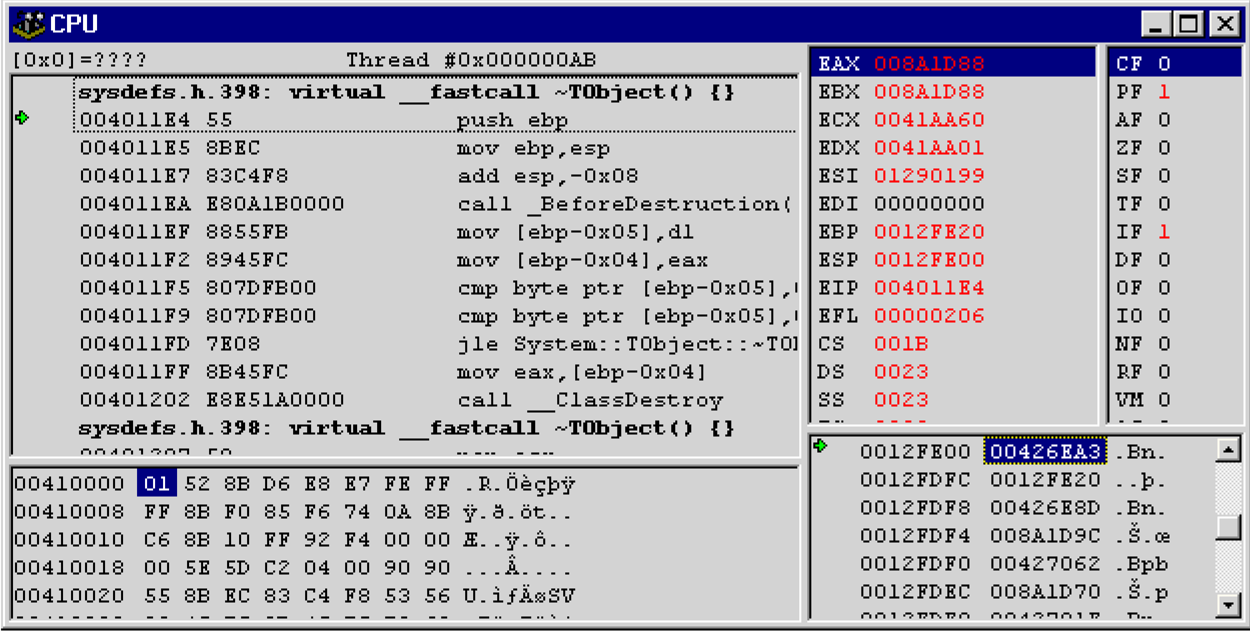
Count
Enter the number of elements in the array you want to change.

Change value

Enter a new value for the data element displayed in an Inspector window.

CPU window

The CPU window consists of five separate panes. Each pane gives you a view into a specific low-level aspect of your running application.



- The <u>Disassembly pane</u> displays the assembly instructions that have been disassembled from your application's machine code. In addition, the Disassembly pane displays the original program source code above the associated assembly instructions.
- The <u>Memory Dump pane</u> displays a memory dump of any memory accessible to the currently loaded executable module. By default, memory is displayed as hexadecimal bytes.
- The <u>Machine Stack Pane</u> displays the current contents of the program stack. By default, the stack is displayed as hexadecimal longs (32-bit values).
- The <u>Registers pane</u> displays the current values of the CPU registers.
- The <u>Flags pane</u> displays the current values of the CPU flags.

Each pane has an individual SpeedMenu that provides commands specific to the contents of that pane.

Opening the CPU window

You can open the CPU window anytime during a debugging session by choosing View|CPU or by choosing CPU View from the Edit window SpeedMenu. In addition, the CPU window opens automatically whenever you step into a function or code that does not contain Borland symbolic debug information. This can occur, for example, when you <u>Step Into</u> a function, when you <u>Attach</u> to a process, or when you start a debugging session with the <u>Just-in-time</u> functionality.

If you choose View|CPU, the debugger positions the Disassembly pane at the location of the execution point. If you open the CPU window from the Edit window SpeedMenu, the debugger positions the Disassembly pane at the instruction that corresponds to the location of the currently selected line in the Edit window.

Resizing the CPU window panes

You can customize the layout of the CPU window by resizing the panes within the window. Drag the pane boarders within the window to enlarge or shrink the windows to your liking.

Disassembly pane

Memory Dump pane

Machine Stack pane

Registers pane

Flags pane

Disassembly pane

The left side of the Disassembly pane lists the address of each disassembled instruction. An arrow to the right of the memory address indicates the location of the current execution point. To the right of the memory addresses, the Disassembly pane displays the assembly instructions that have been disassembled from the machine code produced by the compiler. If you make the Disassembly pane wide enough, the debugger displays the instruction opcodes following the listing of the instruction memory addresses.

If you are viewing code that has been linked with a symbol table, the debugger displays the source code that is associated with the disassembled instructions.

The Disassembly pane supports the following keyboard commands:

- Press Ctrl+N to set the instruction pointer (the value of EIP register) to the beginning of the statement that you have highlighted in the Disassembly pane. Note that this is not the same as stepping through the instructions; the debugger does not execute any instructions that you might skip.
- Press Ctrl+LeftArrow and Ctrl+RightArrow to shift the starting point of the display up or down one byte. Beware that changing the starting point of the display in the Disassembly pane changes where the debugger begins disassembling the machine code.
- The debugger displays dashes if you view a program memory location in which nothing is loaded.

The Disassembly pane SpeedMenu

The Disassembly pane has the following SpeedMenu commands:

- Run To Current
- Set PC To Current
- Toggle Breakpoint
- Go to Address
- Go to Current PC
- Follow jump <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Show previous address
- Go to source
- Change Thread

Run to Current

The Run To Current command lets you run your program at full speed to the instruction that you have selected in the Disassembly pane. After your program is paused, you can use this command to resume debugging at a specific program instruction.

Set PC to current

The Set PC to Current command changes the location of the program counter (the value held in the EIP register) to the currently highlighted line in the Disassembly pane. When you resume program execution in the debugger, it starts at the new address. This command is useful when you want to skip certain machine instructions.

- Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.

Toggle Breakpoint

When you choose Toggle Breakpoint, the debugger sets a "simple" breakpoint at the instruction which you have selected in the Disassembly pane. A simple breakpoint has no conditions, and the only action is that it will pause the program's execution.

If a simple breakpoint exists on the selected instruction, then Toggle Breakpoint will delete the breakpoint at that code location.

Go to address

The Go to Address command prompts you for a new area of memory to display in the Code, Dump, or Machine Stack panes of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with `0x`.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+LeftArrow and Ctrl+RightArrow to shift the starting point of the display up or down one byte.

Go to current PC

This command positions the Disassembly pane at the location of the current program counter (the location indicated by the EIP register). This location indicates the next instruction to be executed by your program.

This command is useful when you have navigated through the Disassembly pane, and you want to return to the next instruction to be executed.

Follow jump <address> into Disassembly pane

This command highlights in the Disassembly pane the destination address of the currently highlighted instruction. Use this command in conjunction with instructions that cause a transfer of control (such as **CALL**, **JMP**, and **INT**) and with conditional jump instructions (such as **JZ**, **JNE**, **LOOP**, and so forth). For conditional jumps, the address is shown as if the jump condition is TRUE. Use the Show Previous Address SpeedMenu command to return to the origin of the jump.

Follow address <address> into Disassembly pane

This command highlights in the Disassembly pane the address of the currently highlighted address. The Show Previous Address SpeedMenu command returns you to the address from where you jumped.

- When you use this command from the Memory Dump pane, set the display to Longs for best results.

Follow address <address> into Memory Dump pane

This command highlights in the Memory Dump pane the address of the currently highlighted address. The Show Previous Address SpeedMenu command returns you to the address from where you jumped.

- When you use this command from the Memory Dump pane, set the display to Longs for best results.

Follow address <address> into Machine Stack pane

This command highlights in the Machine Stack pane the address of the currently highlighted address. The Show Previous Address SpeedMenu command returns you to the address from where you jumped.

- When you use this command from the Memory Dump pane, set the display to Longs for best results.

Show previous address

This command restores the CPU window to the display it had before you issued the last Follow Address command. The Follow Address commands are found on the SpeedMenus of the Disassembly pane, the Machine Stack pane, and the Memory Dump pane of the CPU window.

Go to source

The Go to Source command activates the Edit window and positions the insertion point at the source code that corresponds to the disassembled instruction selected in the Disassembly pane. If there is no corresponding source code (for example, if you're examining Windows kernel code), this command has no effect.

Change thread

Opens the **Change Thread** dialog box. Select the thread you want to debug from the threads listed on the process hierarchy.

If you choose a new thread from the CPU window, all panes in the window reflect the state of the CPU for that thread. Open multiple instances of the CPU window for low-level debugging of different threads.

Memory Dump pane

The Memory Dump pane displays the raw values contained in addressable areas of your program. The display is broken down into three sections: the memory addresses, the current values in memory, and an ASCII representation of the values in memory.

The Memory Dump pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is an 8-byte hexadecimal listing of the values contained at that location in memory. Each byte in memory is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

The format of the memory display depends on the format selected with the Display As SpeedMenu command. If you choose one of the floating-point display formats (Floats or Doubles), a single floating-point number is displayed on each line. The Bytes format (default) displays 8 bytes per line, Words displays 4 words per line, and Longs displays 2 long words per line.

- You can press Ctrl+LeftArrow and Ctrl+RightArrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

The Memory Dump pane SpeedMenu

The Memory Dump pane has the following SpeedMenu commands:

- Go to Address
- Display As
- Change Thread
- Follow address <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Follow address <address> into Machine Stack pane
- Show previous address
- You can change the values of memory displayed in the Memory Dump pane by pressing the Ins key and typing into the display (when you press Insert, the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing program memory values; even small changes in program values can have disastrous effects on your running program.

Display as

Use the Display As command to format the data that's listed in the Dump or Machine Stack pane of the CPU window. You can choose any of the data formats listed in the following table:

Data type	Display format
Bytes	Displays data in hexadecimal bytes
Words	Displays data in 2-byte hexadecimal numbers
Longs	Displays data in 4-byte hexadecimal numbers
Floats	Displays data in 4-byte floating-point numbers using scientific notation
Doubles	Displays data in 8-byte floating-point numbers using scientific notation

Machine Stack pane

The Machine Stack pane displays the raw values contained in the your program stack. The display is broken down into three sections: the memory addresses, the current values on the stack, and an ASCII representation of the stack values.

The Machine Stack pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is a 4-byte listing of the values contained at that memory location. Each byte is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory; non-printable values are represented with a period.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

The format of the memory display depends on the format selected with the Display As SpeedMenu command. If you choose one of the floating-point display formats (Floats or Doubles), a single floating-point number is displayed on each line. The Bytes format displays 4 bytes per line, Words displays 2 words per line, and Longs (the default) displays 1 long word per line.

- You can press Ctrl+LeftArrow and Ctrl+RightArrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

The Machine Stack pane SpeedMenu

The Machine Stack pane has the following SpeedMenu commands:

- Go to Address
- Go to Top Frame
- Go to Top of Stack
- Display As
- Change Thread
- Follow address <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Follow address <address> into Machine Stack pane
- Show previous address
- You can change the values of memory displayed in the Machine Stack pane by pressing the Ins key and typing into the display (when you press Insert, the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing program memory values; even small changes in program values can have disastrous effects on your running program.

Go to top frame
Positions the insertion point in the Machine Stack pane at the address of the frame pointer (the address held in the EBP register).

Go to top of stack
Positions the insertion point in the Machine Stack pane at the address of the stack pointer (the address held in the ESP register).

Registers pane

The Registers pane displays the contents of the CPU registers of the 80386 and greater processors. These registers consist of eight 32-bit general purpose registers, six 16-bit segment registers, the 32-bit program counter (EIP), and the 32-bit flags register (EFL).

After you execute an instruction, the Registers pane highlights in red any registers that have changed value since the program was last paused.

The Registers pane SpeedMenu

The Registers pane has the following SpeedMenu commands:

- Increment Register
- Decrement Register
- Zero Register
- Change Register
- Show Old Registers/Show Current Registers
- Change Thread
- You can change the values of memory displayed in the Registers pane by pressing the Ins key and typing into the display (when you press Insert, the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing register values; even small changes can have disastrous effects on your running program.

Increment register

Increment Register adds 1 to the value in the currently highlighted register. This lets you test “off-by-one” bugs by making small adjustments to the register values.

Decrement register

Decrement Register subtracts 1 from the value in the currently highlighted register. This lets you test "off-by-one" bugs by making small adjustments to the register values.

Zero register

The Zero Register command sets the value of the currently highlighted register to 0.

Change register

Lets you change the value of the currently highlighted register. Tthis command opens the **Change Register** dialog box where you enter a new value. You can make full use of the expression evaluator to enter new values. Be sure to precede hexadecimal values with `0x`.

Show old registers/Show current registers

This command toggles between Show Old Registers and Show Current Registers. When you select Show Old Registers, the Registers pane displays the values which the registers had before the execution of the last instruction. The menu command then changes to Show Current Registers, which changes the display back to the current register values.

Flags pane

The Flags pane shows the current state of the flags and information bits contained in the 32-bit register EFL. After you execute an instruction, the Flags pane highlights in red any flags that have changed value since the program was last paused.

The processor uses the following 14 bits in this register to control certain operations and indicate the state of the processor after it executes certain instructions:

Letters in pane	Flag/bit name	EFL register bit number
CF	Carry flag	0
PF	Parity flag	2
AF	Auxiliary carry	4
ZF	Zero flag	6
SF	Sign flag	7
TF	Trap flag	8
IF	Interrupt flag	9
DF	Direction flag	10
OF	Overflow flag	11
IO	I/O privilege level	12 and 13
NF	Nested task flag	14
RF	Resume flag	16
VM	Virtual mode	17
AC	Alignment check	18

The Flags pane SpeedMenu

The Flag pane has the following SpeedMenu commands:

- Toggle Flag
- Change Thread
- You can change the values of memory displayed in the Flags pane by pressing the Ins key and typing into the display (when you press Insert, the insertion point in the pane shrinks to highlight a single binary value in memory). Be extremely careful when changing flag values; changes here can have disastrous effects on your running program.

Toggle flag

The flag and information bits in the Flags pane can each hold a binary value of 0 or 1. This command toggles the selected flag or bit between these two binary values.

Call Stack window

The Call Stack window displays the sequence of function calls that brought your program to its current state. It deciphers all active functions and their argument values and displays them in a readable format.

The most recently called function displays at the top of the list, followed by its caller and the previous caller to that. The list continues to the first function in the calling sequence, which displays at the bottom of the list.

- Functions called from .DLLs and Windows kernel code also display in the Call Stack window even though they might not have symbolic names associated with them.

The Call Stack window is particularly useful for returning to where a function was called. This is helpful if you accidentally Step Into a function that you meant to Step Over.

The Call Stack window SpeedMenu

The Call Stack window has the following SpeedMenu commands:

- Change Thread
- View Source
- Edit Source

View Source
Displays a non-active Edit window so you can view your source code at the line where the selected function was called (or located on the current execution point if you select the currently running function). If the source file is not currently loaded, the IDE opens a new Edit window.

Edit Source

Displays an active Edit window with the insertion point positioned on the line in your source code where the selected function was called (or positioned at the current execution point if you select the currently running function). If the source file is not currently loaded, the IDE opens it in a new Edit window. Double-click on the function in the Call Stack to achieve the same results.

Returning to where a function was called

The Call Stack window displays the sequence of function calls that brought your program to its current state and the arguments passed to each function.

The Call Stack is particularly useful if you accidentally Step Into code that you wanted to Step Over. To step back and then resume stepping where you originally intended:

1. Choose View|Call Stack to open the Call Stack window.
2. Double-click the call that called the function you stepped into by mistake. It will be the second call from the top in the Call Stack.

   The Call Stack view disappears and the Edit window becomes the active window with your cursor positioned at the location of the call.
3. Move the cursor in the Edit window to a position completely past the call.
4. Choose Run to Cursor on the Edit window SpeedMenu.

Changing the color of the execution point

To change the color of the execution point,

1. Choose Options|Environment.
2. Select Syntax Highlighting and choose Customize.
3. From the Element list, select Execution point.
4. Select the background (BG) and foreground (FG) colors you want.

Debugging class member functions

If you use classes in your programs, you can still use the integrated debugger to step through the member functions in your code. The debugger handles member functions the same way it would step through functions in a program that is not object-oriented.

- If you define a member function inline, then you should check Out-of-line inline funcions to facilitate debugging the inline function.

Debugging external code

If you link external code into your program, you can Step over or Step into that code if the .OBJ file you link in contains debugging information.

You can debug external code written in any language, including C, C++, Object Pascal, and assembly language. As long as the code meets all the requirements for external linking and contains full Borland symbolic debugging information, the integrated debugger can step through it and display the source in an Edit window. If the external code does not contain Borland debug information, you can still step through the code using the CPU window.

Debugging dynamic-link libraries

The debugger automatically loads the symbol table for all .DLLs. This occurs when the program starts or when the program explicitly loads a .DLL (in response to a LoadLibrary call). Because of this, there is no special procedure you need to follow to set breakpoints or to step into .DLLs.

General protection exceptions

If a general protection exception (GP exception) occurs while running your program in the IDE, your program halts and a General Protection Exception dialog box appears. If you choose OK to close the dialog box, the debugger displays the code that is responsible for the GP exception in the Edit window. The execution point marks the offending code.

When this situation occurs, to prevent your program from crashing you must:

1. Choose Debug|Terminate Process..
2. Correct the error that caused the problem before you run your program again.

Although the debugger can catch most GP exceptions, it might fail to catch all of them, depending on the cause of the exception.

Compile-time error

An error that violates a rule of C or C++ syntax. Borland C++ cannot compile your program unless it contains valid C or C++ statements.

The most common causes of compile-time (syntax) errors are

- typographical mistakes
- omitted semicolons
- references to undeclared variables
- wrong number or type of arguments passed to a function
- wrong type of values assigned to a variable

Execution point

The execution point marks the next code statement or assembly instruction to be executed by the debugger. In the Edit window, the execution point is marked by a highlighted line. The Disassembly pane in the CPU window marks the execution point with a green arrow glyph to the left of the address display.

The EIP (program counter) CPU register holds the value that marks the current memory location of the execution point.

Expression

Part of a statement that consists of constants, variables, and data structures combined with operators.

As you debug your code, watching, evaluating, and inspecting operate at the level of expressions. Almost anything you can use as the right side of an assignment statement can be used as a debugging expression.

Logic error

An error in design or implementation that does not prevent your program from compiling, but causes it to not do what you intended. For example, logic errors can result in variables with incorrect or unexpected values, graphic images that display incorrectly, or code that does not execute when expected.

Logic errors are often hard to track down because the IDE cannot find them automatically. The Borland C++ IDE, however, includes debugging features, collectively referred to as the integrated debugger, that can help you locate logic errors.

Run-time error

An error that does not prevent your program from compiling, but causes it to fail with an error when you try to run it. Your program contains legal statements, but the statements do not execute properly.

For example, if your program attempts to open a file that does not exist or tries to divide by zero, the operating system detects the situation and it stops your program from running.

Symbol table

A list of the identifiers used by the debugger to track all variables, constants, types, and function names used in a program. Each executable program and .DLL has its own symbol table.

Warning

A message that appears in the Message window that does not stop your code from compiling, but indicates areas you might want to examine for problems. For example, a warning can alert you to code that

- is inefficient
- is not portable
- violates the ANSI standard

Watch

An expression that lets you track the value of a variable or expression while you step through your code. Use the Watches window to view watches.

As you step through your program, the watch value changes as your program updates the values of the symbols contained in the watch expression.

Using breakpoints

See also

Using a breakpoint is similar to using the Run to Cursor command in that the program runs at full speed until it reaches a certain point. But, unlike Run to Cursor, you can have multiple breakpoints and you can choose to stop at a breakpoint only under certain conditions. Once your program's execution is paused, you can use the debugger to examine the state of your program.

The IDE keeps track of all your breakpoints during a debugging session and associates them with your current project. You can maintain all your breakpoints from a single Breakpoints window and not have to search through your source code files to look for them.

Debugging With Breakpoints

When you run your program from the IDE, it will stop whenever the debugger reaches the location in your program where the breakpoint is set, but before it executes the line or instruction. The line that contains the breakpoint (or the line that most closely corresponds to the program location where the breakpoint is set) appears in the Edit window highlighted by the execution point. At this point, you can perform any other debugging actions.

Setting Breakpoints After Program Execution Begins

While your program is running, you can switch to the debugger (just like you switch to any Windows application) and set a breakpoint. When you return to your application, the new breakpoint is set, and your application will pause or perform a specified action when it reaches the breakpoint.

Adding breakpoints

See also

You can add a breakpoint the following ways:

To set an unconditional breakpoint on a line in your source code

Use one of the following methods:

- Place the insertion point on a line in an Edit window and choose Toggle|Breakpoint from the Edit window SpeedMenu. or press F5 (default keyboard setting).
- Click the gutter in an Edit window next to the line where you want to set a breakpoint.

To set an unconditional breakpoint on a machine instruction

1. Highlight a machine instruction in the Disassembly pane in the CPU window.
2. Choose Toggle Breakpoint on the SpeedMenu or press F5 (default keyboard setting).

To set a conditional breakpoint on a line or machine instruction

1. Place the insertion point on a line in an Edit window or highlight a line in the Disassembly pane of the CPU window.
2. Choose Debug|Add breakpoint or choose Add Breakpoint from the SpeedMenu.
3. Complete the information on the Add Breakpoint dialog box.
4. Click the Advanced button to display the Breakpoint Conditions/Action Options dialog box.
5. Either

- Supply the conditions and action settings you want. See Creating conditional breakpoints.
- Specify option set in the Options input box.

To set other types of breakpoints

1. Choose Debug|Add breakpoint (or press F5 in the default keyboard setting) from anywhere in the IDE or choose Add Breakpoint from the SpeedMenu in an active Edit or Breakpoint window, or the Disassembly pane of the CPU window.
2. Select a breakpoint type on the Add Breakpoint dialog box and supply any additional information associated with the type of breakpoint selected.
3. Either

- Click OK to set an unconditional breakpoint.
- Click the Advanced button to display the Breakpoint Conditions/Action Options dialog box. See Creating conditional breakpoints.

To view the breakpoints set in your code

Choose View|Breakpoint to display the Breakpoints window.

Creating conditional breakpoints

See also

Use a conditional breakpoint when you want the debugger to activate a breakpoint only under certain conditions. For example, you may not want a breakpoint to activate every time it is encountered, especially if the line containing the breakpoint is executed many times before the actual occurrence in which you are interested. Likewise, you may not always want a breakpoint to pause program execution. In these cases, use a conditional breakpoint.

To set a conditional breakpoint

1. Choose Debug|Add breakpoint to open the Add Breakpoint dialog box.
2 Select a breakpoint type and supply the applicable information.
3. Click Advanced to display the Breakpoint Conditions/Action Options dialog box.
4. Click Expr. True and enter an expression that tells the debugger when to trigger the breakpoint. If the condition is not met, the debugger ignores the breakpoint along with any of its actions.
5. If you want the breakpoint to activate only while a specific thread is in process, click Thread ID and enter a Thread ID. Otherwise leave the Thread ID unchecked.
6 If you want the debugger to activate a breakpoint only after it has been reached a certain number of times, click Pass count and enter the number of passes. Otherwise, your program will pause every time the breakpoint is activated.
7. If you want program execution to pause when the breakpoint is activated, click Break (the default). Otherwise, your program will not pause when the debugger activates the breakpoint.
8 If you want the debugger to perform various actions when the breakpoint activates, use the Actions. settings. Otherwise, click OK.

Removing breakpoints

See also

You can remove a breakpoint the following ways:

From an Edit window

- Double-click the gutter in an Edit window next to the line that contains the breakpoint you want to remove.

From an Edit window or the Disassembly pane of the CPU window

1. Place the insertion point on the line or highlight the instruction where the breakpoint is set.
2. Choose Toggle Breakpoint from the SpeedMenu.

From the Breakpoints window

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select one or more breakpoints.
3. Choose Remove Breakpoint(s) from the SpeedMenu.

- To select multiple breakpoints in the Breakpoints window, hold down the Shift or Ctrl key as you select each breakpoint.

Disabling and enabling breakpoints

See also

Disable a breakpoint when you prefer not to activate it the next time you run your program, but want to save it for later use. The breakpoint remains listed in the Breakpoints window and available for you to enable when you want.

To enable or disable a breakpoint

1. Choose View|Breakpoint to open the Breakpoints window.
2. Click the checkbox next to the breakpoint to enable it or clear the checkbox to disable it.

To disable or enable selected breakpoints

1. In the Breakpoints window, hold down the Shift or Ctrl key as you select each breakpoint.
2. Choose Enable/Disable Breakpoints from the SpeedMenu.

To use a breakpoint to disable or enable a group of breakpoints

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Click Options to open the Breakpoint Conditions/Action Options dialog box.
3. Click Enable Group or Disable Group and enter a group name.

Inspecting and editing code at a breakpoint

Even if a breakpoint is not in your current Edit window, you can quickly locate it in your source code.

To view the code where a breakpoint is set

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select a breakpoint.
3. Choose View Source on the Breakpoints window SpeedMenu.

The source code displays in an Edit window at the breakpoint line and the Breakpoints window remains active. If the source code is not currently open in an Edit window, the IDE opens a new Edit window.

To edit the code where a breakpoint is set

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select a breakpoint.
3. Choose Edit Source from the Breakpoints window SpeedMenu.

The source code displays in an active Edit window with your cursor positioned on the breakpoint line, ready for you to edit. If the source code is not currently open in an Edit window, the IDE opens a new Edit window.

Resetting invalid breakpoints

See also

A breakpoint must be set on executable code; otherwise, it is invalid. For example, a breakpoint set on a comment, a blank line, or a declaration is invalid. A common error is to set a breakpoint on code that is conditionalized out.

If you set an invalid breakpoint and run your program, the debugger displays an Invalid Breakpoint dialog box.

To reset an invalid breakpoint

1. Close the Invalid Breakpoint dialog box.
2. Open the Breakpoints window.
3. Find the invalid breakpoint and delete it.
4. Set the breakpoint in a proper location and continue to run your program.

- If you ignore the Invalid Breakpoint (by dismissing the dialog box) and then choose Run, the IDE executes your program, but does not disable the invalid breakpoint.

Using Breakpoint groups

The integrated debugger lets you group breakpoints together so you can enable or disable them with a single breakpoint action.

To create a breakpoint group

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Enter a name in the Group input box.

To disable or enable a group of breakpoints

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Click Options to open the Breakpoint Conditions/Action Options dialog box.
3. Click Enable Group or Disable Group and enter a group name.

- To remove a breakpoint from a group, select the group name and press Delete.

Using breakpoint option sets

To quickly specify the behavior of one more breakpoints as you create or modify them, store breakpoint settings in an option set.

To create an option set

1. Choose Debug|Breakpoint options to open the Breakpoint Conditions/Action Options dialog box.
2. Enter the conditions and actions. See Creating conditional breakpoints.
3. Click Add.
4. Enter a name in the dialog box that displays and click OK.

- You can also create an option set when you create or edit a breakpoint.

To associate a breakpoint with an option set

Enter an option set in Names on the Breakpoint Conditions/Action Options dialog box when you add or edit a breakpoint.

To delete an option set

1. Choose Debug|Breakpoint options to open the Breakpoint Conditions/Action Options dialog box.
2. Select an Option set and click Delete.

Changing breakpoint options

See also

To change the conditions and actions of a breakpoint

1. Choose View|Breakpoint to open the Breakpoints window.
2. Double-click on a breakpoint or choose Edit Breakpoint from the SpeedMenu.
3. Change the option set in the Options input box on the Edit Breakpoint dialog box.

   or

   Supply new information as described in Creating conditional breakpoints.

Option set

A set of options that stores settings specified on the Breakpoint Conditions/Action dialog box that you can use to control the behavior of one or more breakpoints. See Using breakpoint option sets.

Changing the color of breakpoint lines

To use colors to indicate if a breakpoint is enabled, disabled, or invalid

1. Choose Options|Environment.
2. Select Syntax Highlighting and choose Customize.
3. From the Element list, select the following breakpoint options you want to change:
   Enabled Break
   Disabled Break
   Invalid Break
4. Select the background (BG) and foreground (FG) colors you want.

Breakpoints window

See also

The Breakpoints window lists all breakpoints currently set in the loaded project (or the file in the active Edit window if a project is not loaded) and contains a tab for each of the following breakpoint types.

- To display the Breakpoints window, choose View|Breakpoint.

How to use the Breakpoints window

The Breakpoints window lets you perform the following actions:

- Click the checkbox beside a breakpoint to enable it or clear the checkbox to disable the breakpoint.
- Double-click on a breakpoint to open the Edit Breakpoint dialog box to change breakpoint settings.
- Choose a command from the Breakpoint Window SpeedMenu.

Information on the Breakpoints window

The Breakpoints window provides the following information about each breakpoint:

- Name of the source code file in which the breakpoint is set.
- Location (such as line number, file name, module, thread ID, or address number) where the breakpoint is set.
- Current state of the breakpoint:

Verified	The breakpoint is legal and validated when the process was loaded.
Unverified	The process has not been loaded since you added the breakpoint.
Invalid	The breakpoint is illegal. The line on which you set the breakpoint does not contain executable code (such as a blank line, comment, or declaration) and the debugger will ignore it.

- Number of times the debugger must reach the breakpoint before activating the breakpoint. This information appears after a breakpoint has been activated. See Pass Count.
- Associated option set and group name as well as the conditions/action options specified. See Creating conditional breakpoints.
- **<– – Last Event Hit** shows the breakpoint last encountered.

Add Breakpoint dialog box

See also

Use this dialog box to create a breakpoint.

Breakpoint type

The options that appear in the middle of the dialog box change according to the breakpoint type selected:

Source	Address	Data Watch	C++ Exception
OS Exception	Thread	Module	

The following options always display on the right side of the dialog box:

Qualifiers

Other

- If you want to set conditions and actions that control breakpoint behavior, click Advanced to open the Breakpoint Conditions/Action Options dialog box.

Qualifiers

Contains the following options:

Program	Causes a breakpoint to activate only in a specific executable (.EXE) program when more than one process is attached to the debugger. Leave it blank if you want a breakpoint to activate in all attached processes.
Module	Causes a breakpoint to activate only when conditions are satisfied within a specific dynamic-link library (DLL.). This option has no effect if you selected Module as the breakpoint type.

Other

Contains the following options:

Options	Indicates the name of the option set that defines breakpoint behavior.
Group	Indicates the name of group to which the breakpoint belongs.

Source breakpoint

See also

Sets a breakpoint on a line in your source code.

File

Indicates the file that contains the source code where the breakpoint is set.

Line #

Indicates the line in the source file on which the breakpoint is set.

- If you select a line of code in an Edit window and choose Add Breakpoint from the SpeedMenu, the debugger completes these settings for you.

Address breakpoint

See also

Sets a breakpoint on a machine instruction.

Offset

Indicates the address of the machine instruction on which the breakpoint is set.

Data Watch breakpoint

See also

Use a Data Watch breakpoint to pause your program when a specific location in memory changes value (when the location is written to). Data Watch breakpoints (also called watchpoints or changed memory breakpoints) let you monitor expressions that evaluate to a specific data object or memory location. Data Watch breakpoints are monitored continuously during your program's execution.

Because the debugger checks the breakpoint conditions after the execution of every line of source code or machine instruction, Data Watch breakpoints are excellent tools for pinpointing code that is corrupting data.

Address

Enter a specific starting address or any symbol (such as a variable or a class data member or method) that evaluates to an address.

- If you enter an address expression that evaluates to a momory location that contains executable code, the Data Watch breakpoint behaves like an Address breakpoint; the breakpoint fires when the code at the specified address is executed.

Length

When entering an address expression, you can also enter a count of the number of objects you want to monitor. For example, suppose you have declared the following C array:

```
int string[81];
```

You can watch for a change in the first ten elements of this array by entering the following item into the Condition Expression input box:

```
&string[0], 40
```

The area monitored is 40 bytes long which equals ten elements in the array (an **int** is 4 bytes).

Using data watch breakpoints

When your program execution encounters an address that contains a Data Watch breakpoint, the condition expression is evaluated before the line of code gets executed. Therefore, carefully consider the placement of Data Watch breakpoints.

- Because the debugger evaluates Data Watch breakpoints after the execution of every machine instruction, setting this type of breakpoint slows the execution of your program. Be moderate with your use of Data Watch breakpoints; use them only when you need to closely monitor the behavior of your program.

C++ exception breakpoint

See also

Sets a breakpoint that pauses your program when it throws or catches a C++ exception.

Type

Specifies the exception data type (such as **int**, **long**, **char**, or a class name) which you want to associate with the breakpoint. If you enter an ellipses (...) into the Type field, the debugger will trap any C++ exception that is thrown or caught by your program.

Stop on Throw

Pauses program execution when an exception is thrown.

Stop on Catch

Pauses program execution when an exception is caught.

Stop on Destructor

Pauses program execution when any object is destroyed (when a destructor is called) after an exception is thrown.

OS Exception breakpoint

See also

Sets a breakpoint that pauses your program when it raises an operating system defined exception.

Exception #

Specifies the integer value assigned to the operating system exception you want to track. Pick from the list that provides the most common OS exceptions or enter one defined in your application.

- Be aware that if you are running Borland C++ on Windows NT, an Illegal Instruction generates the Exception 0xC000001E, and not 0xC000001D. Because of this, if you want to trap an Illegal Instruction exception, be sure to use the value 0xC000001E on Windows NT systems (Windows 95 systems can correctly use the 0xC000001D value).

Exception Through

Use Exception Through to specify a range of exceptions to track. Choose a value from the drop-down list (or enter your own value) to specify the higher value in the range of the exceptions you want to track. When you specify an Exception Through value, the debugger tracks all the exceptions between and including the values specified in the Exception # and Exception Through fields. Leave this field blank to trap the single exception specified in the Exception # input box.

- To track all exceptions, specify 0 in the Exception # field and 0xFFFFFFFF in the Exception Through field. To facilitate this entry, these numbers appear in the drop-down lists.

Thread breakpoint

See also

Sets a breakpoint on a thread create event.

Thread

Specifies the thread ID on which the breakpoint is set. Program execution pauses whenever the thread is created.

- To obtain a thread ID, choose View|Process.

Module breakpoint

See also

Sets a breakpoint on a module (or .DLL) load event.

Module Name

Specifies the name of a DLL in which the breakpoint is set. The debugger activates the breakpoint each time the DLL is loaded.

Opens the Breakpoint Conditions/Action Options where you specify settings that control the behavior of one or more breakpoints.

Breakpoint Conditions/Action Options

See also

Use this dialog box to

- specify settings that control the behavior of one or more breakpoints, such as the conditions under which a breakpoint is activated and the type of actions that take place when it does
- enable and disable breakpoint groups

To display this dialog box, use any of the following methods:

- Choose Debug|Breakpoint Options.
- Choose Debug|Add Breakpoint and click the Advanced button on the Add Breakpoint window.
- Choose View|Breakpoint and double-click a breakpoint listed in the Breakpoints window. Then click the Advanced button on the Edit Breakpoint window.

The Breakpoint Conditions/Action Options dialog box contains the following options:

Names	Lists the names of Option Sets that have been created.
Conditions	Provides settings that determine when and where a breakpoint is activated.
Actions	Provides settings that determine what actions take place when a breakpoint is activated.

Names (Breakpoint Conditions/Action Options)

Lists the names of existing option sets. Use the checkbox next to each option set to enable or disable it.

For example, if you clear the checkbox next to an option set called `MyOptionSet`, the debugger ignores its settings and all breakpoints that use this option set behave like unconditional breakpoints. To reactivate the breakpoint settings in `MyOptionSet` so that they will used by the debugger, click its checkbox.

Conditions (Breakpoint Conditions/Action Options)

This group of settings determines when and where a breakpoint is activated:

Expr. True	Each time the debugger encounters the breakpoint, it evaluates an expression to determine if the breakpoint should activate.
Thread ID	Activates a breakpoint only while a specific thread is in process.
Pass Count	Indicates the number of times the debugger encounters the breakpoint line before it activates.

- Click Add or Delete to create or remove an option set.

Expr. True (Breakpoint Conditions/Action Options)

Enter the expression you want to evaluate each time the debugger reaches the breakpoint. If the expression becomes true (nonzero) when the breakpoint is encountered, the debugger activates the breakpoint and carries out any actions specified for it. You can enter a Boolean expression that, for instance, tests if a value falls within a certain range or if a flag has been set.

For example:

If you enter the expression

```
x == 1
```

the debugger activates the breakpoint only if *x* has been assigned the value 1 at the time the breakpoint is encountered.

If you enter the expression

```
x > 3
```

and select Break, when the debugger reaches the breakpoint, your program pauses if the current value of *x* is greater than 3. Otherwise, the breakpoint is ignored.

Thread ID (Breakpoint Conditions/Action Options)

Programs written for 32-bit operating systems consist of one or more executable threads. You can set a breakpoint on a specific thread, even though the code at the breakpoint location is shared by multiple threads. Unless specified, a breakpoint is set for all program threads.

Use this option to prevent a breakpoint from activating unless a specified thread is running. When the debugger reaches the breakpoint, it will not be activated unless the thread is in process. If you leave this option blank, the breakpoint may activate while any thread is in process.

- To obtain a valid thread ID, choose View|Process.

Pass Count (Breakpoint Conditions/Action Options)

This option includes the following settings:

Up to Specifies the number of times you want debugger to reach the breakpoint before it is activated.

Current Shows the actual number of times the debugger has reached the breakpoint so far. You can change this setting in cases where undesired side effects of have resulted from an Eval Expr expression specified as a breakpoint action.

- If you specify an Expr. True expression, the pass count determines how many times the expression must be satisfied before it is activated.

Unconditional breakpoint example

Suppose Break is checked, and in the Pass Count box you enter `2`. In this case, your program does not stop until the second time the debugger reaches the breakpoint.

Conditional breakpoint example

Suppose Break is checked, plus you enter the expression `x>3` and in the Pass Count box you enter `2`. In this case, your program does not stop until the second time the debugger reaches the breakpoint when the value of x is greater than 3.

Actions (Breakpoint Conditions/Action Options)

See also

This group of options lets you specify the actions you want carried out each time the breakpoint is activated:

Break	Pauses program execution
Stop Log	Stops posting debugger generated messages
Start Log	Starts posting debugger generated messages
Log Expr	Displays the value of an expression in the message window
Eval Expr	Evaluates an expression
Log Message	Displays a message in the message window
Enable Group	Reactivates a group of breakpoints
Disable Group	Disables a group of breakpoints

Break (Breakpoint Conditions/Action Options)

Click Break (the default) to pause program execution when the debugger activates the breakpoint. Clear this checkbox if you do not want your program to pause at the breakpoint.

Stop Log (Breakpoint Conditions/Action Options)

Stops displaying debugger messages in the Run time Tab of the Message window when the breakpoint is activated.

Start Log (Breakpoint Conditions/Action Options)

Starts displaying debugger messages in the Run time Tab of the Message window when the breakpoint is activated.

Log Expr (Breakpoint Conditions/Action Options)

Click Log Exp if you want to display the value of an expression in the Run time tab of the Message window. Then, enter the expression in the input box next to it. The debugger logs the value each time the breakpoint activates. Use this option when you want to output a value each time you reach a specific place in your program — this technique is known as instrumentation.

For example, you can place a breakpoint at the beginning of a routine and set it to log the values of the routine arguments. Then, after running the program, you can determine where the routine was called from, and if it was called with erroneous arguments.

- When you log expressions, be careful of expressions that unexpectedly change the values of variables or data objects (side effects).

Eval Expr (Breakpoint Conditions/Action Options)

Click Eval Expr if you want the breakpoint to evaluate an expression. Then, enter an expression in the input box next to it. For best results, use an expression that changes the value of a variable or data object (side effects).

By “splicing in” a piece of code before a given source line, you can effectively test a simple bug fix; you do not have to go through the trouble of compiling and linking your program just to test a minor change to a routine.

- You cannot use this technique to directly modify your compiled program.

Log Message (Breakpoint Conditions/Action Options)

Click Log Message if you want the breakpoint to display a message in the Run time tab of the Message window when the breakpoint is activated. Then, enter the text of the message in the input box next to it.

Enable Group (Breakpoint Conditions/Action Options)

Click Enable Group if you want the breakpoint to reactivate a group of breakpoints that have been previously disabled. Then, enter a group name in the input box next to it.

Disable Group (Breakpoint Conditions/Action Options)

Click Disable Group if you want the breakpoint to disable a group of breakpoints. Then, enter a group name in the input box next to it.

- When a group of breakpoints is disabled, the breakpoints are not erased, they are simply hidden from the debugger until you enable them.

Opens the Add Breakpoint Conditions/Action Option Set Name dialog box where you can create a breakpoint option set.

Deletes the selected breakpoint option set. If you have a breakpoint that uses a deleted option set, the integrated debugger treats that breakpoint as an unconditional breakpoint.

Add Conditions/Action Option Set Name

Enter a name for the option set and click OK to create a new set of breakpoint options. Then enter your selections using the Breakpoint Conditions/Action Options dialog box.

Edit Breakpoint

Use this dialog box to modify an existing breakpoint. The options that appear on left side of the dialog box change according to the breakpoint type selected.

Source	Address	Data Watch	C++ Exception
OS Exception	Thread	Module	

The following options always display on the right side of the dialog box:

Qualifiers

Other

- If you want to set conditions and actions that control breakpoint behavior, click Advanced to open the Breakpoint Conditions/Action Options dialog box.

Breakpoint types

The integrated debugger provides several types of breakpoints:

Source	Address	Data Watch	C++ Exception
OS Exception	Thread	Module	

Breakpoint

A place in your code where you want the program to pause or perform an action so that you can examine the current values of program variables and data structures. Breakpoint behavior falls into two categories:

- Unconditional (or simple). The breakpoint is activated whenever the debugger reaches the line in your source or the machine instruction where you set the breakpoint.
- Conditional. The breakpoint is activated only when it satisfies the conditions specified on the Breakpoint Conditions/Action Options dialog box or saved in a breakpoint options set.

The integrated debugger provides several types of breakpoints:

Source	Address	Data Watch	C++ Exception
OS Exception	Thread	Module	

- A breakpoint must be set on a line or program location that contains executable code. For example, you cannot set a breakpoint on a blank line, comment, or declaration.

Building Applications with AppExpert

See also

AppExpert lets you create ObjectWindows-based Windows applications with features such as a tool bar, a status bar, a menu structure, online Help, and MDI windows. You can also select options to support printing, print preview, and document/view. AppExpert works with Resource Workshop, ObjectWindows classes, and the IDE's project manager to form a visual approach to application generation.

Creating applications with AppExpert consists of four steps:

1. Use AppExpert to define the user interface and application features and to generate the code.
2. Use ClassExpert to add classes and event handlers, to implement virtual functions, to navigate to existing class source code, and automate classes. ClassExpert can also associate Resource Workshop objects (such as menus and dialog boxes) with classes or handlers.
3. Use Resource Workshop to edit or add resources.
4. Use the project tree to build your application.

AppExpert always creates the following files for each application:

- A database file for the AppExpert source (.APX) that ClassExpert uses
- A main header file (.H)
- A main source file (.CPP)
- A project file (.IDE)
- A resource header file (.RH)
- A resource script file (.RC)

Depending on which options you choose, AppExpert can create the following files:

- Help source files (.RTF)
- A Help project file (.HPJ)
- Icon and bitmap files (.ICO and .BMP)

Steps for Creating an Application with AppExpert

To create an AppExpert application:

1. Start the IDE and choose File|New|AppExpert. A dialog box appears.
2. Type a name for your project file. By default, most generated files (including the .EXE) are derived from the target name (for example, `<targetname>.CPP`). Select a path where you want the AppExpert project and source files to be stored. AppExpert will create the directory if it does not already exist. Click OK. The AppExpert Application Generation Options dialog box appears.
3. The Application Generation Options dialog box contains a list of topics on the left and a brief description of the topic on the right. You can change options in the dialog box to customize the type of application you want generated.
4. Click the Generate button at the bottom of the Options dialog box.
5. The Generate dialog box appears, asking you to confirm that you want the code generation for your application to begin. Click Yes to generate the code for your application. While AppExpert is generating your application, a message box displays the current status.
6. The project window appears, listing some of the files required for your application (files for bitmaps, icons, and help text are not displayed). You can use ClassExpert to modify your application or you can build it first. To build your application, choose either Project|Make All or Project|Build All.

- The application generates and builds faster, if you uncheck unneeded options.
- With AppExpert, you choose your application options once, then generate the code. After you generate the code and resources, you can edit and add to them, but you cannot go back to AppExpert and change options. For example, if you use AppExpert to generate an application that does not contain a status line and then decide to add one. You will need to write code to add that functionality.

AppExpert Application Generation Options

See also

The AppExpert Application Generation Options affect the code generation for your application.

Subtopics

Application

Main Window

MDI Child/View

Application options

Use the AppExpert|Application options to define the specific characteristics of the application you want to generate.

Options
Window Model

Customize Application button
Use the Customize Application button to open the group (if necessary) and go to the first option in the first subtopic available for the Application section of the AppExpert options.

Generate button
If you simply want to use the default options, you can generate your application now by pressing the Generate button.

Subtopics
Basic Options

Advanced Options

OLE 2 Options

Code Gen Control

Admin Options

Application | Window Model

The Application|Window Model options to determine which model to use for your application and how application objects will be handled.

MDI (Multiple Document Interface)
The MDI option sets the style of your application to follow the Multiple Document Interface (MDI) model. This model causes child windows to be constrained within the boundaries of the application window.

SDI (Single Document Interface)
The SDI option sets the style of your application to follow the Single Document Interface (SDI) model. This model supports child windows that can exist outside of the constraints of the application window without being clipped.

Dialog Client
The Dialog Client option sets the style of your application so that the main (and only) window of your application is a dialog window, making the client area of the application a dialog box.

Document/View
The Document/View option determines whether your application supports the Document/View model for handling application objects. The Document/View model breaks data handling into two discrete parts: data storage and control, in the document class (derived from TDocument), and data display and manipulation, in the view class (derived from TView).

You can use the Document/View option with either SDI, MDI, or Dialog Client applications.

Application | Basic Options | Features to Include

Use the Application|Basic Options|Features to Include options to specify which additional user interface elements your application supports.

Dockable Toolbar
Turn on the Dockable Toolbar option to include a dockable toolbar in your application.

Status Line
Turn on the Status Line option to place a status line at the bottom of your application's main window. The code generated will display help hints in the status line when menu items are highlighted.

Recently Used Files List
Turn on the Recently Used Files List option to include a list of the four most recently accessed files under the File menu of your application.

Registry Support
Turn on the Registry Support option to register your application’s icons and document types with the system registry when the application is run.

Drag/Drop
Turn on the Drag/Drop option to have code generated so that your application will support drag-and-drop actions from File Manager and Explorer.

Printing
Turn on the Printing option if you want your application to support printing-related activities. The code generated will handle the Print Setup, Print Preview, and Print commands.

Mail Support
Turn on the Mail Support option to include support for Common Message Call (CMC) mail in your application.

Help File Support
Use the Help File Support option to determine whether help source and project files are generated for the project.

Help File Name
Use Help File Name to specify the name of the help file associated with your application. This is also the name of the node associated with the help file in your AppExpert project.

Application | Basic Options

The Application|Basic Options options define the general characteristics of the application you are going to generate, such as the initial state of the application's main window, and the generation of help files, as well as the directories where files will be stored.

If you simply want to use the default options, you can generate your application now by pressing the Generate button.

Target Name

Use Target Name to define the name of the AppExpert target you want to create within the IDE project.

Base Directory

Use the Base Directory to set the base directory path from which all of the AppExpert project directories will be located.

Features to Include

Use the Features to Include options to specify which additional user interface elements your application supports.

Application | Advanced Options

The Application|Advanced Options options define some specific characteristics of the application you are going to generate, such as the initial state of the application's main window, and the type of controls the application will use.

Application Startup State
Use the Application Startup State options to set the initial state of the application's main window.

Control Style
Use the Control Style options to determine which type of controls the application will use.

Target Name

Use Application|Basic Options|Target Name to name the AppExpert target that will be used to build your application. This name is the basis for the default names of other elements in your project (e.g., header files, class database, application class, and source files).

The default name is the same as the IDE project. You can use this name or enter your own.

Base Directory

Use Application|Basic Options|Base Directory to define the main project directory. All other directories associated with the project are placed relatively to this directory.

You can enter a directory by entering it yourself or press the Browse button to select one from the Select Directory dialog box.

The default base directory is the one you choose in the New AppExpert Project dialog box. The name of this directory is passed to the Project Manager for the new AppExpert target.

Help File Support

If you select the Application|Basic Options|Include Help File Support option, AppExpert will generate RTF source files and a Help project file when it generates your application. The Help project file will be added to the Project Manager project and automatically built with the target application.

This Help file contains placeholder text for the menus and menu options present in the generated application.

Help File Name

Use Application|Basic Options|Help File Name to specify the name of the help file associated with your application. This file name identifies the source and Help project files.

The default value for the file name is the same as the application project. You can also enter your own file name.

Application | Advanced Options | Application Startup State

Use the Application | Advanced Options | Application Startup State options to set the initial state of the application's main window.

Normal (sizeable)

Turn on the Normal option to set the application window to start in a default size. This is the default setting.

Minimized (iconic)

Turn on the Minimized option to set the application window to start in a minimized state (as an icon on the Windows desktop).

Maximized (entire screen)

Turn on the Maximized option to set the application window to fill the entire Windows desktop when it starts running.

Application | Advanced Options | Control Style

Use the Application | Advanced Options | Control Style options to determine which type of controls the application will use.

Standard Windows
Turn on the Standard Windows option to specify that the application will use standard Windows controls. This is the default setting.

Borland BWCC
Turn on the Borland BWCC option to specify that the application will use the Borland custom control style.

MS Control 3D
Turn on the MS Control 3D option to specify that the application will use the three-dimensional Windows controls provided by CTL3DV2.DLL and CTL3D32.DLL.

Application | OLE 2 Options

Use the Application | OLE 2 Options options to set the behavior of your OLE 2 application.

OLE 2 Container Options

Choose whether or not you want your application to be an OLE 2 and OCX container application. AppExpert only allows MDI and SDI applications which use the DocView model to be containers.

OLE 2 Server Options

Choose whether you want your application to be an OLE 2 Server EXE, DLL, or not a server at all. AppExpert only allows MDI and SDI applications which use the DocView model to be servers.

Enable Automation in the Application

Select the Enable Automation in the Application option if you want your application to be an OLE 2 automation server.

Server ID

If you set your application to be an OLE 2 Server, use the Server ID field to specify the identifier for your application.

Application | Code Generation Control

Use the Application |Code Generation Control options to control various aspects of the code generation process.

Target Name

Displays the name of the project as defined in Basic Options|Target Name.

Base Directory

Displays the base directory for the project as defined in Basic Options|Base Directory.

Source Directory

Use Source Directory to specify the directory where the source files for the application will be placed during code generation.

Header Directory

Use Header Directory to specify the directory where the header files for the application will be placed during code generation.

Main Source File

Use Main Source File to name the main application source file. The default filename is the name of the project: `<TargetName>app.CPP`. AppExpert will parse and shorten the project name if it is longer than eight characters, unless the Use Long File Names option is enabled.

Main Header File

Use Main Header File to name the main application header file. The default filename is the name of the project: `<TargetName>app.H`. AppExpert will parse and shorten the project name if it is longer than eight characters, unless the Use Long File Names option is enabled.

Application Class

Use Application Class to name the class that AppExpert will derive from TApplication. The default class name is `T<TargetName>App`.

About Dialog Class

Use About Dialog Class to name the class that AppExpert will derive from TDialog. The default class name is `T<TargetName>AboutDlg`.

Use Long File Names

The Use Long File Names option determines whether or not the source files of your application are generated using long file names.

- Some installations of Novell NetWare do not support long file names.

Comments

Use the Comments options to specify how the generated code is documented.

Source Directory

See also

Use Application|Code Generation Control|Source Directory to specify the directory where the source files for the application will be placed during code generation. This path is relative to the directory specified as the Base Directory. If an absolute path is specifed, it is converted to a path relative to the Base Directory.

You can enter a directory or press the Browse button to select one from the Select Directory dialog box. The default value for the Source Path is the current directory, which is the directory specified in Base Directory.

Header Directory

See also

Use Application|Code Generation Control|Header Directory to specify the directory where the header files for the application will be placed during code generation. This path is relative to the directory specified as the Base Directory. If an absolute path is specifed, it is converted to a path relative to the Base Directory.

You can enter a directory or press the Browse button to select one from the Select Directory dialog box. The default value for the Header Path is `".\"` which causes source files to be placed in the current directory.

Application | Code Gen Control | Comments

Use the Application|Code Generation Control|Comments options to specify how much of the generated code is documented.

Terse
Turn on the Terse option to sparsely comment the generated code.

Verbose
Turn on the Verbose option to heavily comment the generated code.

Application | Administrative Options

Use the Application|Administrative Options to specify identifying information that will be placed in a comment block at the beginning of all of the generated project files. Some of the information is also displayed in the About... dialog box of the application.

Version Number

Use Version Number to provide the version number for the project. This value will be displayed in the application's "About . . ." dialog box. The default version number is "1.0".

Copyright

Use Copyright to provide the copyright information for your application. This value will be displayed in the application's "About . . ." dialog box and will be placed in all of the comment blocks generated by AppExpert.

Description

Use Description to provide a description of your application. This value will be displayed in the application's "About . . ." dialog box. The default value is the name of the project.

Author

Use Author to provide the name of the programmer who generated the source code. This will be placed in all of the comment blocks generated by AppExpert.

Company

Use Company to provide the name of the programmer's company. This will be placed in all of the comment blocks generated by AppExpert.

Main Window summary

Use the AppExpert|Main Window to control the appearance and operation of the main window of the generated application:

Window title	Specifies the name that will be displayed as the caption for the application's main window.
Background color	Use Background Color to determine the background of the application's main window.
Customize main window	Use the Customize Main Window button to open the group (if necessary) and go to the first option in the first subtopic available for the Main Window section of the AppExpert options.
Generate	If you want to simply use the default options, you can generate your application now by pressing the Generate button.

Subtopics

Basic Options

SDI Client

MDI Client

Dialog Client

Main Window | Background Color

The Main Window|Background Color options determine the background color of the application’s main window.

Use default color

The Use Default Color options causes no code to be generated to set the main window background color. In this way, the color used will be the current default for the system.

Use system color constant

The Use System Color Constant option sets the background color of the main window to a specified constant. Choose the constant from the list box provided.

Use specified color

The Use Specified Color option sets the background color of the main window to a specified color. Choose the color from the Color dialog.

Main Window | Basic Options

Use the Main Window|Basic Options to control the general appearance your application's main window. The Window Styles let you specify its border style and whether it contains system and control menus.

Caption
Turn on the Caption radio button to create a single, thin border and a title bar where a caption can be displayed.

Border
Turn on the Border option to get a single, thin border without a caption for the application's main window.

Max Box
Turn on the Max Box option to add a maximize button to the right side of the application's main window caption.

Min Box
Turn on the Min Box option to add a minimize button to the right side of the application's main window caption.

Vertical Scroll
Turn on the Vertical Scroll option to add a vertical scroll bar to the right side of the application's main window.

Horizontal Scroll
Turn on the Horizontal Scroll option to add a horizontal scroll bar to the bottom of the application's main window.

System Menu
Turn on the System Menu option to add a system menu button on the left side of the application's main window caption.

Visible
Turn on the Visible option to make the application's main window visible. This is the default setting. When this option is off, the WS_VISIBLE style is changed to NOT WS_VISIBLE.

Disabled
Turn on the Disabled option to disable the application's main window.

Thick Frame
Turn on the Thick Frame option to place a double border around the application's main window. If this option if off, users cannot resize the main window.

Clip Siblings
Turn on the Clip Siblings option to protect the siblings of the application's main window. Painting is restricted to that window.

Clip Children
Turn on the Clip Children option to protect child windows from being repainted by the application's main window.

Color dialog box

Use the Color dialog box to select the background color you want for the background of your main application window.

Main Window | SDI Client Window

Use the Main Window|SDI Client Window options to define the class that represents the client area of the Single Document Interface main window. These options are only applicable if you select the Single Document Interface option from Application|Window Model options.

Client/View Class
Use Client/View Class to name the class of the SDI client area window or view.

Document Class
Use Document Class to name the class of the default document. If the Doc/View option in the Application|Window Model options is selected, the default value is TFileDocument; otherwise, Document Class is blank.

Description
Use Description to describe the class of files associated with the Files of Type List in Windows common file dialog boxes. The default value is "`All Files (*.*)`". This value serves as the description for the value entered in Filters.

Filters
Use Filters to list wildcard file specifications (separated by semicolons) for the file names you want the application to recognize. This value is passed to Windows common file dialog boxes to filter files displayed in the list box of those dialog boxes. The default value is "`*.*`". The values entered in Description serves as the explanation of this value.

Default Extension
Use Default Extension to specify the default filename extension. This value is passed to Windows common file dialog boxes and added to filenames without extensions. The default value is "`TXT`".

Class Name
Use Class Name to specify the name AppExpert uses for the class derived from Client/View Class that represents the client area of the SDI frame window. The default value is `T<ProjectName><Client/ViewClass>` (without the leading "T").

Source File
Use Source File to name the source file that will store the implementation of the class named in Client/View Class. The default value is `<ClassName>.CPP`. AppExpert will parse and shorten the class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Header File
Use Header File to name the header file that will store the definition of the class named in Client/View Class. The default value is `<ClassName>.H`. AppExpert will parse and shorten the class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Client/View Class

Use Main Window|SDI Client|Client/View Class to name the class of the SDI client area or view. The interpretation of this value depends on whether you select the Doc/View option in the Application|Window Model options.

If the Doc/View option is not selected, Client/View Class selects the class of the client window; otherwise, Client/View Class selects the class of the view of the default document/view.

	Doc/View
On	**Off**
TEditView (default)	TEditFile (default)
TListView	TListBox
TWindowView	TWindow

This value is automatically mapped to the Doc/View options. For example, if you turn off the Doc/View option, TListView is switched to TListBox. Conversely, if you select the Doc/View option, TListBox changes to TListView.

Main Window | MDI Client Window

Use the Main Window|MDI Client Window options to describe the class that defines the client window of the Multiple Document Interface main window. These options are only applicable if you select the Multiple Document Interface option from Application|Window Model options.

Client Class

Use Client Class to specify the name AppExpert uses for the class derived from TMDIClient that represents the client area of the MDI frame window. The default value is `T<TargetName><ClientClass>`.

Source File

Use Source File to name the source file that will store the implementation of the class named in Client Class. The default value is `<ClientClass>.CPP`. AppExpert will parse and shorten the client class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Header File

Use Header File to name the header file that will store the definition of the class named in Client Class. The default value is `<ClientClass>.H`. AppExpert will parse and shorten the client class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Main Window | Dialog Client Window

Use the Main Window|Dialog Client Window options to describe the characteristics of an application where the main window is a dialog window.

Client Class

Use Client Class to specify the name AppExpert uses for the class derived from TDialog that represents the main window of the application. The default value is `T<TargetName><ClientClass>`.

Source File

Use Source File to name the source file that will store the implementation of the class named in Client Class. The default value is `<ClientClass>.CPP`. AppExpert will parse and shorten the project name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Header File

Use Header File to name the header file that will store the definition of the class named in Client Class. The default value is `<ClientClass>.H`. AppExpert will parse and shorten the class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Dialog ID

Use Dialog ID to specify the identifier associated with the dialog that functions as the main window of your application.

Include a Menu Bar

Select Include a Menu Bar if you want a menu bar on your main window.

MDI Child/View Options

Use the AppExpert|MDI Child/View options to define the class that will define a default child window or document/view of the Multiple Document Interface application. These options are only applicable if you select the Multiple Document Interface option from Application|Window Model options.

Select...	To name the...
MDI Child Class	class derived from TMDIChild that represents the frame of the default MDI child windows. The default value is `T<TargetName>MDIChild`.
Source File	source file that will store the implementation of the class named in MDI Child. The default value is `<MDIChildClass>.CPP`. AppExpert will parse and shorten the child class name if it is larger than eight characters, unless the Use Long File Names option is enabled
Header File	header file that will store the definition of the class named in Client Class. The default value is `<MDIChildClass>.H`. AppExpert will parse and shorten the child class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Click the...	If you want to...
Customize Child and View button	open the group and go to the first set of options available for the MDI Child/View section of the AppExpert options. This is the same as double-clicking on the group name in the left pane when the section of the outline is collapsed.
Generate button.	simply use the default options and generate your application now.

Subtopics
Basic Options

MDI Child/View | Basic Options

Use the MDI Child/View|Basic Options to define the default child window of an MDI application.

MDI Client/View Class
Use MDI Client/View Class to name the class of the default MDI view.

Document Class
Use Document Class to name the class of the default document. If the Doc/View option in the Application|Window Model options is selected, the default value is TFileDocument.; otherwise, Document Class is blank.

Description
Use Description to describe the class of files associated with the Files of Type List in Windows common file dialog boxes. The default value is "`All Files (*.*)`". This value serves as the description for the value entered in Filters.

Filters
Use Filters to list wildcard file specifications (separated by semicolons) for the file names you want the application to recognize. This value is passed to Windows common file dialog boxes to filter files displayed in the list box of those dialog boxes. The default value is "`*.*`". The values entered in Description serves as the explanation of this value.

Default Extension
Use Default Extension to specify the default filename extension. This value is passed to Windows common file dialog boxes and added to filenames without extensions. The default value is "TXT".

Class Name
Use Class Name to specify the name AppExpert uses for the class derived from the MDI Client/View Class that represents the client area of the MDI frame window. The default value is `<ProjectName><ClientViewClass>` (without the leading "T").

Source File
Use Source File to name the source file that will store the implementation of the class entered in Class Name. The default value is `<ClassName>.CPP`. AppExpert will parse and shorten the project name if it is larger than eight characters, unless the Use Long File Names option is enabled.

Header File
Use Header File to name of the header file that will store the definition of the class entered in Class Name. The default value is `<ClassName>.H`. AppExpert will parse and shorten the class name if it is larger than eight characters, unless the Use Long File Names option is enabled.

MDI Client/View Class

Use MDI Child/View|Basic Options|MDI Client/View Class to name the class of the MDI client area or view. The interpretation of this value depends on whether you selected the Doc/View option in the Application|Window Model options.

If the Doc/View option is not selected, MDI Client/View Class selects the class of the client window; otherwise, MDI Client/View Class selects the class of the view of the default document/view.

	Doc/View
On	**Off**
TEditView (default)	TEditFile (default)
TListView	TListBox
TWindowView	TWindow

This value is automatically mapped to the Doc/View options. For example, if you turn off the Doc/View option, TListView is switched to TListBox. Conversely, if you select the Doc/View option, TListBox changes to TListView.

Modify Options dialog box

Use the Modify Options dialog box to change the current target name or to enable long file names. The OK button for this dialog is only enabled when the target name and the use long file names option are not in conflict.

New Target Name

Use New Target Name to define the name of the AppExpert target you want to create within the IDE project.

Use Long File Names

The Use Long File Names option determines whether or not the source files of your application are generated using long file names.

Select Directory dialog box

Use the Select Directory dialog box to choose a directory for input boxes that require valid directory names and paths.

Directories
The Directories list box displays the directories available on the current drive.

Selected Path
The Selected Path display box shows the currently selected directory path.

New Subdirectory
Use the New Subdirectory input box to create a new subdirectory from the Selected Path.

Select Directory dialog box

Use the Select Directory dialog box to choose a directory for input boxes that require valid directory names and paths.

Directories
The Directories list box displays the directories available on the current drive.

Selected Path
The Selected Path display box shows the currently selected directory path.

New Subdirectory
Use the New Subdirectory input box to create a new subdirectory from the Selected Path.

Drives
Use the Drives list box to select a drive from the list of drives currently accessible from your system.

Use the Customize button to open the group (if necessary) and go to the first option in the first subtopic available for the current section of the AppExpert options. From there, you can set specific options for the AppExpert application you are going to build.

Use the Generate button to start AppExpert generating the code for your application. AppExpert displays the Generate dialog box and prompts for your confirmation before starting to generate code.

Use the Browse button to display the Select Directory dialog box in which you can select a directory.

Use the Network button to access drives and directories available on your network.

Generate dialog box

The Generate dialog box is displayed when you press the Generate button. It prompts you for confirmation before starting to generate code.

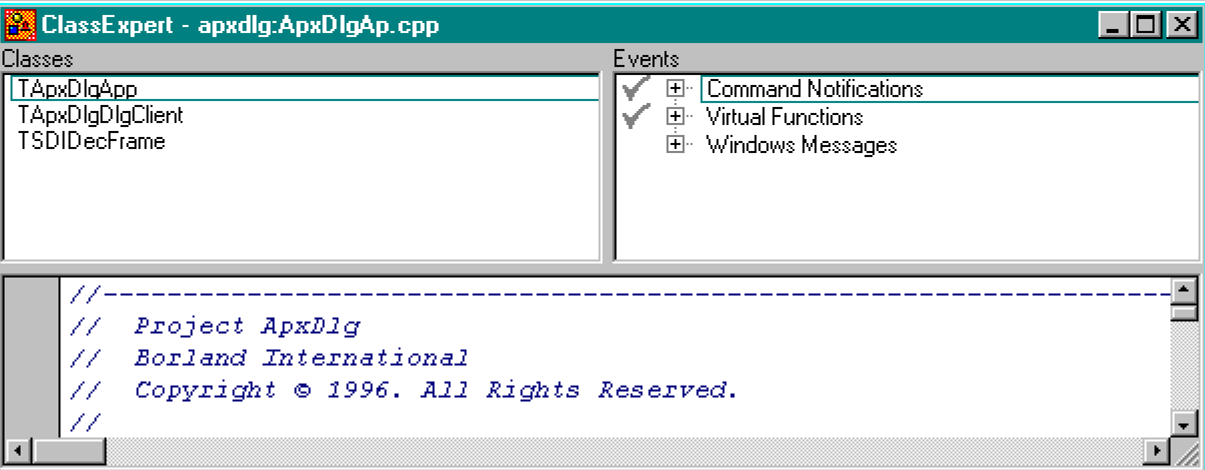
- Click *Yes* to begin building your AppExpert application.
- Click *No* if you do not want your AppExpert application built. You will be returned to the AppExpert|Application options dialog box.

Before starting code generation, AppExpert verifies that options you have selected are valid. If any options are not valid, AppExpert returns you to the AppExpert Application Generation Options dialog box.

Managing your classes with ClassExpert

See also

ClassExpert lets you create new classes, edit and refine the implementation of classes, and navigate through the source code for existing classes in your AppExpert applications. You can use ClassExpert with Resource Workshop to associate classes to resources (for example, associating a TDialog class to a dialog resource). ClassExpert displays virtual functions and events for existing classes and checks the ones implemented in your application. You can also use ClassExpert to instantiate and automate classes in your AppExpert project.



To start ClassExpert:

1. Open an AppExpert project file by opening the AppExpert .IDE file (choose File|Open).
2. Double-click the AppExpert target node (ClassExpert is the default viewer for AppExpert targets), or choose View|ClassExpert or click the toolbar button (the button is not visible by default). ClassExpert appears, listing the classes and their implementation for your application.

The ClassExpert window contains three panes, each with a specific function:

Classes	displays the classes used in the application
Events	displays the events handled by the selected class
Edit	lets you view and edit the source associated with the selected class

Using Rescan to rebuild your project database

Rescan is a special project tool that examines all the source code listed in your AppExpert project (.IDE file) and updates or rebuilds the project's database (the .APX file) according to what it finds in the source code. Rescan looks for special markers in the source code to reconstruct the AppExpert database file and then starts Resource Workshop to reconstruct information about project resources. If Rescan is successful, the original project database file is renamed to *.~AP and a new database file is created; otherwise, the original database is left as *.APX.

You can use Rescan to:

- Delete a class
- Move a class from one source file to another
- Rename a class, handler, instance variable, or dialog ID
- Import a class from another AppExpert project
- Rebuild a lost or damaged project database (*.APX) file

Deleting a class

To delete a class:

1. Remove the class source file from the IDE project by selecting the source node, right-clicking the node, and choosing Delete Node. If the class shares a source file with other classes, delete the code related to the class from the source file and delete references to the class from other source files; otherwise, you will get errors during compilation.
2. Select the AppExpert target in the project, right-click it, then choose Special|Rescan. Rescan scans the source files listed as dependencies for the AppExpert target. Resource Workshop scans and updates the resource files. When Rescan is complete, you will return to the updated project file where you can either build your application or use ClassExpert.

   You can add the deleted class back to the project by adding the class source file (and its associated header file as a dependent node under the class source file) as a dependent of the AppExpert target (use Add Node from the SpeedMenu), then rescanning.
3. Look at the ClassExpert Classes pane in ClassExpert to see that the class is now omitted from the list.

Moving a class

To move a class from one source file to another:

1. Move (cut and paste) the code or definition of the class to the new source or header file. Be certain to also copy the comments generated by AppExpert or else ClassExpert will be unable to locate the correct location in the new source or header files where the class is referenced or defined.

   Add the new source file if is not in the project (and its associated header file as a dependent node under the class source file) as a dependent node of the AppExpert target (use Add Node from the SpeedMenu). If the moved class was in its own source file, you can delete the now-empty source file from the project.
2. Select the AppExpert target in the project, right-click it to view the SpeedMenu, then select Special| Rescan. When Rescan is finished, it returns you to the project window in the IDE.

Renaming an AppExpert element

To rename a class, event handler function, instance variable, or dialog ID:

1. Use the IDE editor to search and replace all occurrences of the original name with the new name. Be sure to check all source files associated with the project (.CPP, .H, .RC, and .RH files).
2. In the project window, select the AppExpert target, right-click it, then choose Special|Rescan. When Rescan is finished, it returns you to the project window in the IDE.

- After following this procedure, check your #include statements, as well as the #ifdef and #define statements that AppExpert created. Verify that none of these have changed as a result of searching and replacing the name of your class within the source files.

Importing a class

To import a class from one AppExpert project to another:

1. Move or copy the source and header file that defines the class to the source and header directory of the other project. All source files for a project must be in the project's source directory (.CPP files) or header directory (.H files). These directories were created when you first generated the AppExpert project. (Unless you specified a different source or header directory, your source and header files will be in the same directory as your project [.IDE] file.)
2. Add the class source file as a dependent node under the AppExpert target in the IDE project (use Add Node from the SpeedMenu).
3. Add the header file that contains the class definition as a dependent node under class source file in the AppExpert project (use Add Node from the SpeedMenu).
4. In the project window, select the AppExpert target, right-click, then choose Special|Rescan.

Rebuilding the .APX database file

To rebuild a lost or damaged database file (the .APX file):

1. Open the project (.IDE) file that contains the AppExpert target and dependent nodes.
2. Select the AppExpert target, right-click it, then choose Special|Rescan from the SpeedMenu. Rescan automatically creates a new database file using markers and the source code from the AppExpert application.

ClassExpert Classes pane

ClassExpert

The Classes pane lists the set of classes known to ClassExpert that are used in the current project. This includes all classes created during code generation and any classes you might have added afterward using ClassExpert.

Selecting a class from this list changes the contents displayed in the Events and Edit panes to show the events and source file associated with the selected class.

Classes pane SpeedMenu

Use the Classes pane SpeedMenu to access frequently used ClassExpert commands.

Add New Class

Automate Class

Delete Automation (automated classes only)

View Document Templates (Document/View applications only)

View Class Info

Edit Dialog (dialog classes only)

Edit Menu

Edit Source

Edit Header

Add New Class

Use the Add New Class command from the Classes pane SpeedMenu to create a new class in your application. Choosing this option displays the Add New Class dialog box which you can use to define the new class you are creating.

Automate Class

Use the Automate Class command from the Classes pane SpeedMenu to expose existing classes in your application to OLE 2 automation. This will cause additional code to be added to your source files to support the automation of the class currently selected in the ClassExpert Classes pane. If the TApplication class is not already exposed, automating a class will automatically automate it.

Delete Automation

Use the Delete Automation command from the Classes pane SpeedMenu to remove the OLE 2 automation code from the class currently selected in the ClassExpert Classes pane.

Choosing this command displays a dialog box warning you that the class will be removed from automation. Click Yes to remove the automation code, or No to cancel the operation.

- This command is only available if the class has already been automated.

View Document Templates

Use the View Document Templates command from the Classes pane SpeedMenu to create, delete, and view document and view pairings. Choosing this option displays the Document Templates dialog box in which you can define the document templates.

This option is only applicable if you selected the Document/View option from the Application|Window Model settings in AppExpert when you set the options for your application.

View Class Info

See also

Use the View Class Info command from the Classes pane SpeedMenu to view a summary of the properties of the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Class Info dialog box.

If the selected class is derived from TFrameWindow, you can also use this option to change the client class and view the window styles and background color.

Edit Source

See also

Use the Edit Source command from the Classes pane SpeedMenu to edit the source file associated with the class currently selected in the ClassExpert Classes pane.

Choosing this option opens the source file in an IDE Edit window (not in the ClassExpert Edit pane) and positions the cursor on the class constructor.

Edit Dialog

See also

Use the Edit Dialog command to edit the dialog template associated with the class currently selected in the ClassExpert Classes pane. Choosing this command starts Resource Workshop and loads the associated dialog resource into the Dialog Editor.

This option is only applicable if the currently selected class is derived from TDialog.

Edit Menu

See also

Use the Edit Menu command to edit the menu resource associated with the class currently selected in the ClassExpert Classes pane. Choosing this command starts Resource Workshop and loads the associated menu resource into the Menu Editor.

Edit Header

See also

Use the Edit Header command from the Classes pane SpeedMenu to edit the header file associated with the class currently selected in the ClassExpert Classes pane.

Choosing this option opens the header file in an IDE Edit window (not in the ClassExpert Edit pane) and positions the cursor on the class definition.

ClassExpert Events pane

ClassExpert

The Events pane contains an outliner list with several categories. The available categories depend on the derivation of the class currently selected in the ClassExpert Classes pane.

These are the categories that can be displayed:

Automation (automated classes only)

Command Notifications

Control Notifications (dialog classes only)

Virtual Functions

Windows Messages

Events pane SpeedMenu

Use the Events pane SpeedMenu to access frequently used ClassExpert commands.

Add Handler

Delete Handler

Edit Menu

For dialog classes (those derived from TDialog), you will get these additional commands:

Add Instance Variable

Delete Instance Variable

Edit Dialog

For Automation events, the following additional Events pane SpeedMenu commands are vailable:

Data	Methods	Properties
Add Data	Add Method	Add Property
Delete Data	Delete Method	Delete Property
View Data	View Method	View Property

Events | Automation

See also

The Events|Automation category lists all of the automation data, methods, and properties for the automated class currently selected in the ClassExpert Classes pane.

Events | Command Notifications

See also

The Events|Command Notifications category lists all of the commands inherited from the base class of the class currently selected in the ClassExpert Classes pane, as well as command events that can be generated by other objects in the application, such as menus. This list is dynamically updated to reflect any changes you might make to objects that generate commands in your application.

Each item in the Command Notifications category expands to two items: Command and Command Enabled. The Command handler causes the associated class to handle the command event, while the Command Enabled handler allows the associated class to control whether the command event is enabled or disabled (grayed).

A checkmark next to an event indicates that it is handled by the class currently selected in the Classes pane.

Events | Control Notifications

See also

The Events|Control Notifications category appears only for classes derived from TDialog. This category lists the identifiers for all of the controls in the dialog associated with the class. Each control identifier expands to a list of messages it can send to or receive from the dialog class.

A checkmark next to an event indicates that it is handled by the class currently selected in the ClassExpert Classes pane. A light gray checkmark indicates that one or more of the child events are handled.

Events | Virtual Functions

See also

The Events|Virtual Functions category lists all of the virtual functions inherited from the base class of the class currently selected in the ClassExpert Classes pane. The contents of this list is fixed for a given class.

A checkmark next to an event indicates that it is handled by the class currently selected in the Classes pane.

Events | Windows Messages

The Events|Windows Messages category lists the set of Windows messages. This set includes:

- Basic Messages
- Other Messages
- Win32 Messages
- OLE 2 Messages

A checkmark next to a message indicates that it is handled by the event currently selected in the Events pane.

Add Handler

Use the Add Handler option from the Events pane SpeedMenu to create a default member function skeleton in the source file for the class currently selected in the ClassExpert Classes pane. The source file will appear in the Edit pane.

- If you add a handler for a Windows message, ClassExpert adds an entry to the response table whose name is defined by default. The function associated with the handler appears in the Edit window with the appropriate default handling.
- If you add a handler for a Virtual Function, the function associated with the handler appears in the Edit window with the appropriate default handling.
- If you add a handler for either Commands or Control Notifications, ClassExpert will prompt you for the function name before adding the entry to the response table.

This option is only applicable if the event currently selected in the Events pane is not already handled by a member function of the selected class. Unhandled events and non-overriden virtual functions will not have a checkmark next to them.

Delete Handler

Use the Delete Handler option from the Events pane SpeedMenu to delete a member function from the event currently selected in the Edit pane.

- The member function is removed, but the code that implements the function is not deleted. You will be warned of this and will need to remove the code. For events, commands, and control notifications, the response table entries are removed.
- This option is only applicable if the currently selected event has a member function associated with it; the handled event will have a checkmark next to it.

Add Instance Variable

Use the Add Instance Variable option from the Events pane SpeedMenu to add an instance variable for the item currently selected in the Events pane.

Choosing this option prompts you for the name of the variable. The type of the variable is fixed and determined by the associated control. ClassExpert then adds the following to your application code:

- In the header file, a structure declaration is added with an entry for the instance variable.
- In the source file,
  1. the instance variable is allocated in the class constructor (this associates the ObjectWindows class with the resource object).
  2. a static instance of the transfer structure is declared.
- This command is only available if:
- The currently selected item is a control notification and does not already have an instance variable associated with it.
- The class currently selected in the ClassExpert Classes pane is derived from TDialog.

Delete Instance Variable

Use the Delete Instance Variable option from the Events pane SpeedMenu to delete the instance variable associated with the item currently selected in the Events pane.

ClassExpert deletes the following from your code:

- The entry from the structure
- The pointer variable in the class declaration
- The allocation of the class variable associated with the resource control in the constructor

If you delete all instance variables from your code, you will have an empty structure and the set transfer buffer call. This information does not affect the rest of your code, so you do not need to delete it manually.

- This command is only available if:
- The currently selected item is a control notification and has an instance variable associated with it.
- The class currently selected in the ClassExpert Classes pane is derived from TDialog.

Instance variables

Instance variables let you handle many controls easily. When you create instance variables, ClassExpert adds a transfer buffer in your code for the controls that use them. This transfer buffer collects information at run time, so that you avoid creating code to check each check box.

Add Data

Use the Add Data command to add automation data to the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Data dialog box where you can enter information about the automation data.

- This command is only available for classes that you have exposed to OLE 2 automation using the Automate Class command.

Delete Data

Use the Delete Data command to remove automation data from the class currently selected in the ClassExpert Classes pane.

- The automation data is removed from the OLE 2 automation symbol table, but the code that implements the data is not deleted. You will be warned of this and will need to remove the code.

This option is only applicable if the currently selected class has automation data associated with it and you have selected a data member in the Events pane.

View Data

Use the View Data command to view the automation data for the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Data dialog box where you can view (but not change) the information about the selected automation data.

This option is only applicable if the currently selected class has automation data associated with it and you have selected a data variable in the Events pane.

ClassExpert Edit pane

ClassExpert

The Edit pane is an Edit window that supports most of the features of the editor in the IDE. It contains the C++ source file associated with the class currently selected in the ClassExpert Classes pane.

You can freely edit the file displayed in the Edit pane, adding, deleting, and modifying code as needed.

Edit Pane SpeedMenu

Use the Edit pane SpeedMenu to access frequently used ClassExpert commands.

Open Source

Go to Help topic

Go to line

Toggle Breakpoint

Run to

Set Watch

Inspect

Use Class

Use Class

This command opens the Use Class dialog box where you choose which class you want to instantiate and tell ClassExpert to generate the code to instantiate the class in your application.

Use Class dialog box

Use the Use Class dialog box to choose the class you want to instantiate in your application.
ClassExpert

- Generates code (embedded in a comment) that instantiates the class
- Adds a header file is be added to the source containing the class defintiion (if it is not already present) file. You will need to change the name of the class to a unique name and remove the comment specifiers.

Example: `// TheAboutDialog *xTheAboutDialog = new TheAboutDialog()`

Class to Instantiate

Use the Class to Instantiate list box to select the class you want to instantiate. By default, all of the classes listed in the ClassExpert Classes pane will be displayed. If Use OWL Base Classes is selected, you will also see all OWL base classes in the list.

Type of Instantiation

Use the Type of Instantiation radio buttons to specify how you want the class instantiated.

Static

Choose Static to have the class instantiated statically.

Dymanic (new/delete)

Choose Dymanic to have the class instantiated dymanically, meaning that the new and delete operators are used.

Use OWL Base Classes

Select Use OWL Base Classes if you want the ObjectWindows base classes displayed in the Class to Instantiate list box.

Automate Data dialog box

The Automate Data dialog box lets you enter and view automation data members. If you chose View Data, you will only be able to view the information in each field and not change it.

Name
Use the Name input box to enter the name associated with the automation data member you are creating.

Type
Use the Type list box to select the type of automation data member you are creating.

Description
Use the Description to enter a description for the automation data member you are creating.

Add Method

Use the Add Method command to expose an automation method to OLE 2 automation in the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Method dialog box where you can enter information about the automation method.

- This command is only available for classes that you have exposed to OLE 2 automation using the Automate Class command.

Delete Method

Use the Delete Method command to remove an exposed method selected in the ClassExpert Classes pane from the OLE 2 automation symbol table.

- The automation method is removed from the OLE 2 automation symbol table, but the code that implements the method is not deleted. You will be warned of this and will need to remove the code.
- This option is only applicable if the currently selected class has an automation method associated with it and you have selected a method in the Events pane.

View Method

Use the View Method command to view the automation method for the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Method dialog box where you can view the information (but not change it) about the selected automation method.

- This option is only applicable if the currently selected class has an automation method associated with it and you have selected a method in the Events pane.

Automate Method dialog box

The Automate Method dialog box is lets you enter and view an automation method. If you chose View Method, you will only be able to view the information in each field and not change it.

Name
Use the Name list box to enter the name associated with the automation method you are creating. The list box will display all of the functions currently defined in your class. You can also enter a the name of a new method.

Return Value
Use the Return Value list box to select the type of the return value for the automation method you are creating.

Parameters
Use the Parameters input box to specify the parameters for the automation method you are creating. The format is `(type1 name1, type2 name2, . . ., typen namen)`. The parentheses can be omitted. You can also leave out the names of the parameters, in which case ClassExpert will provide the names `p1`, `p2`, etc.

Description
Use the Description to enter a description for the automation method you are creating.

Add Property

Use the Add Property command to expose a property to OLE 2 automation of the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Property dialog box where you can enter information about the automation property.

Delete Property

Use the Delete Property command to remove an exposed automation property from the class currently selected in the ClassExpert Classes pane.

- The automation property is removed from the OLE 2 automation symbol table, but the code that implements the property is not deleted. You will be warned of this and will need to remove the code.
- This option is only applicable if the currently selected class has an automation property associated with it and you have selected a property in the Events pane.

View Property

Use the View Property command to view the automation property for the class currently selected in the ClassExpert Classes pane. Choosing this command displays the Automate Property dialog box where you can view information about the selected automation property.

This option is only applicable if the currently selected class has an automation property associated with it and you have selected a property in the Events pane.

Automate Property dialog box

The Automate Property dialog box lets you enter and view an automation property. If you chose View Property, you will only be able to view the information in each field and not change it.

Name
Use the Name input box to enter the name associated with the automation property you are creating.

Type
Use the Type list box to select the type of automation property you are creating.

Get
Use the Get checkbox to specify whether the property can be read. Selecting this option causes a C++ function to be created that returns the value of the property.

Set
Use the Set checkbox to specify whether the property can be written to. Selecting this option causes a C++ function to be created that passed a new value to the property.

Description
Use the Description input box to enter a description for the automation property you are creating.

Add New Class dialog box

Use the Add New Class dialog box to define the properties of a new class you are adding to your AppExpert application. Most of these properties cannot be changed after their values have been determined, except by directly modifying the source code that controls those properties.

Base class

Use the Base Class combo box to select the base class from which the new class will be derived.

Class came

Use Class Name to name the new class you are creating.

Source file

Use Source File to name the source file that will contain the implementation of the new class. The default value is `<ClassName>.CPP`. AppExpert will parse and reduce the class name to eight characters unless Use long file names is enabled.

Header file

Use Header File to name the source file that will contain the definition of the new class. The default value is `<ClassName>.H`. AppExpert will parse and reduce the class name to eight characters unless Use long file names is enabled.

Show all OWL classes

Choose the Show All OWL Classes option if you want all of the available ObjectWindows classes displayed in the Base Class combo box. This allows you to select non-GUI classes for your application that are not derived from TEventHandler or TWindow.

Use long file names

The Use Long File Names option determines whether or not the source files for this class are generated using long file names. Note that some installations of Novell NetWare do not support long file names.

Resource ID

Use Resource ID to specify the dialog template associated with the dialog class named in Base Class. You can select an existing dialog identifiers from the list or enter your own. If you enter a new identifier, a new default dialog template will be created and assigned that identifier.

- This option is available only if the value of Base Class is TDialog.

Client Class

Use Client Class to select the class of the client area of the new frame window. This is the only value that can be changed after you create the class.

- This option is available only if the value of Base Class is TFrameWindow or a class derived from TFrameWindow.

Set Window Properties button

Press the Set Window Properties button to set the properties for the new frame window. Pressing this button displays the Window Styles dialog box. This option is applicable for all windows derived from TWindow, except for those derived from TDialog.

Class Info dialog box

Use the Class Info dialog box to display the properties of the class you have selected in the ClassExpert Classes pane.

Value	Meaning
Base class	The base class of the currently selected class.
Class name	The currently selected class.
Source file	The source file that contains the implementation of the currently selected class.
Header file	The source file that contains the definition of the currently selected class.
Resource ID	The Dialog template associated with the Base class.

- The Resource ID appears only if the Base class is TDialog.

Client class	The class of the client area of the frame window. This is the only value that can be changed after you create the class.

- The Client Class appears only if the Base class is TFrameWindow or a class derived from TFrameWindow.

Window Styles dialog box

Use the Window Styles dialog box to set the properties for a new window class.

Window Title
Use Window Title to specify the name to be displayed as the caption of the new window class.

Background Color
The Background Color options determine the background color of the new window class.

Caption
Turn on the Caption radio button to create a single, thin border and a title bar where a caption can be displayed.

Border
Turn on the Border option to get a single, thin border without a caption for the application's main window.

Maximize Box
Turn on the Maximize Box option to add a maximize button to the right side of the application's main window caption.

Minimize Box
Turn on the Minimize Box option to add a minimize button to the right side of the application's main window caption.

Vertical Scroll
Turn on the Vertical Scroll option to add a vertical scroll bar to the right side of the application's main window.

Horizontal Scroll
Turn on the Horizontal Scroll option to add a horizontal scroll bar to the bottom of the application's main window.

System Menu
Turn on the System Menu option to add a system menu button on the left side of the application's main window caption.

Visible
Turn on the Visible option to make the application's main window visible. This is the default setting. When this option is off, the WS_VISIBLE style is changed to NOT WS_VISIBLE.

Disabled
Turn on the Disabled option to disable the application's main window.

Thick Frame
Turn on the Thick Frame option to place a double border around the application's main window. If this option if off, users cannot resize the main window.

Clip Siblings
Turn on the Clip Siblings option to protect the siblings of the application's main window. Painting is restricted to that window.

Clip Children
Turn on the Clip Children option to protect child windows from being repainted by the application's main window.

Background Color options

The Background Color options determine the background color of the new window class.

Use Default Color

The Use Default Color option uses the current default for the system for the background color.

Use System Color Constant

The Use System Color Constant option sets the background color to a specified constant. Choose the constant from the list box provided.

Use Specified Color

The Use Specified Color option sets the background color to a specified color. Choose the color from the Color dialog.

Document Templates dialog box

Use the Document Templates dialog box to define which document and view pairs to use in your application.

Existing Document Templates

Existing Document Templates lists the current set of document/view pairs available for your application.

Add button

Click Add to open the Add New Document Template dialog where you create a new document template. The new template is added to the Existing Document Templates.

Delete button

Click Delete to remove the document/view pair currently selected in the Existing Document Templates.

Document Class

Document Class displays the document class used by the currently selected Document Template.

Doc Styles button

Click Doc Styles to display the Document Styles dialog box where you set the styles of the Document/View pair.

View Class

View Class displays the view class used by the currently selected Document Template.

Description

Description displays the description used by the currently selected Document Template.

Filters

Filters displays the wildcard file specifications (separated by semicolons) for the file names you want the currently selected Document Template to recognize. This value is passed to Windows common file dialog boxes to filter files displayed in the list box of those dialog boxes.

Default Extension

Default Extension displays the the default file-name extension for the currently selected Document Template. This value is passed to Windows common file dialog boxes and added to file names without extensions.

Modify button

Click Modify to open the Modify Document Template dialog box where you can edit the currently selected Document Template.

Add New/Modify Document Template dialog box

Use the Add New Document Template and Modify Document Template dialog boxes to add or modify a Document/View pair to be used by your application.

Document Class

Use Document Class to name the class of the document. The values possible include any class derived from TDocument.

Doc Styles button

Press the Styles button to display the Document Styles dialog box, which lets you set the styles of the Document/View pair.

View Class

Use View Class to name the class of the view. The values possible include any class derived from TView.

Description

Use Description to describe the class of files associated with the Files of Type List in Windows common file dialog boxes. This value serves as the description for the value entered in Filters.

Filters

Use Filters to list wildcard file specifications (separated by semicolons) for the file names you want this Document Template to recognize. This value is passed to Windows common file dialog boxes to filter files displayed in the list box of those dialog boxes. The values entered in Description serves as the explanation of this value.

Default Extension

Use Default Extension to specify the default file-name extension. This value is passed to Windows common file dialog boxes and added to file names without extensions.

Document Styles dialog box

Use the Document Styles dialog box to define the document style for the document/view pair defined in the Document Templates dialog box.

dtAutoDelete	Deletes the document when the last view is deleted.
dtNoAutoView	Does not automatically create the default view type.
dtSingleView	Provides only a single view for each document.
dtAutoOpen	Opens a document upon creation
dtUpdateDir	Updates the directory with the dialog directory.
dtHidden	Hides the template from the user's selection.
dtSelected	Indicates the last selected template.
dtReadOnly	Checks the Read Only check box when the dialog is created. (OFN_READONLY)
dtOverwritePrompt	When the Save As dialog is displayed, asks the users if its okay to overwrite the file. (OFN_OVERWRITEPROMPT)
dtHideReadOnly	Hides the Read Only check box. (OFN_HIDEREADONLY)
dtPathMustExist	Lets the user enter only existing file names in the File Name entry field. If an invalid file name is entered, causes a warning message to be displayed. (OFN_PATHMUSTEXIST)
dtFileMustExist	Lets the user enter only existing file names in the File Name entry field. If an invalid file name is entered, causes a warning message to be displayed. (OFN_FILEMUSTEXIST)
dtCreatePrompt	Prompts the user before creating a document that does not exist. (OFN_CREATEPROMPT)
dtNoReadOnly	Returns the specified as writeable. (OFN_NOREADONLYRETURN)

Project management

See also

As an application grows in size and complexity, it becomes dependent on various intermediate files. Often, source files need to be compiled with different compilers and different sets of compiler options. Even a simple Windows program can have resource scripts and C++ source files, with each file type requiring different compilers and different compiler settings.

As your project complexity increases, the need increases for a way to manage the different components in the project. Looking at the files that make up a project, you can see that a project combines one or more *source files* to produce a single *target file*. While target files are usually executable .DLLs or .EXEs, source files cover a broader range of file types, including .C, .CPP, .RC, .ASM, and .DEF files. Additionally, many source files have *autodependant files* (files that are automatically included by the source), such as C header files. In larger projects, you are likely to find several targets with scores of sources.

Project management is the organization and management of the source and target files that make up your project. In addition, project management encompasses how and when you employ different tools to translate the source files into your project target files.

Project management tools

See also

Borland C++ provides several tools to help you manage your application projects.

Project Manager	The Project Manager is the main tool for managing projects in Borland C++. Use the View|Project command to access the Project Manager, a collapsible/expandable, hierarchical display of the files in your project.
Project menu	The Project menu provides commands to open and close projects, add a new target to a project, and make, build, or compile targets.
Options Hierarchy	The View Options Hierarchy command (located on the Project Tree window SpeedMenu) opens a dialog box that lets you set options for individual project nodes.
Node attributes	The Edit Node Attributes command (located on the Project Tree window SpeedMenu) lets you control how each node is handled by the Project Manager.
Tools	Use the Options|Tools command to install, delete, or modify the tools that you use in your projects.
TargetExpert	TargetExpert opens when you create a new project or add a new target node to an existing project. TargetExpert makes available the appropriate platform, model, and library choices based on the type of target you select.

About the Project Manager

See also

The Project Manager visually organizes all the files in your project in a hierarchy diagram known as the *project tree*. The project tree represents each file in your project as a *node* on the tree. The project tree is divided into discrete levels where each level contains a single target node. Indented below each target node are the target's *dependencies*-the files used to build the target.

When you first open the IDE, the Project Manager is displayed in the upper right corner of the IDE.

Creating a project

When you begin to write a new application, the first step is to create a new project to organize your application files. The File|New|Project command opens the New Target dialog box, which you use to set up your application target type, libraries, and so on.

Depending on the options you choose with TargetExpert, the Project Manager creates a project with the appropriate default nodes. For example, if you create the a new 32-bit OWL program, the Project Manager fills out the project tree with the default .CPP, .DEF, and .RC files. If you turn on Show Run-time Nodes (Options|Evnironment|Project View), the project tree also reflects the .OBJ and .LIB files that are automatically set by the Project Manager.

You can convert a project from previous versions of C++. You can also add a Borland C++Builder makefile to your project, as well as convert your .IDE file into a .MAK file for Borland C++Builder.

Working with nodes

By manipulating the various node types and their settings, you can manage most project details and make changes to settings at a project or node level, assign tools to nodes, add target (output file) nodes, or adjust dependencies by moving nodes within the project.

When you choose a node, the Project Manager invokes the default action on the node. The default action for a specific node depends upon the attributes that are set on the node. For example, a viewer could be invoked if the node can be edited, or a translator might compile or assemble the node.

Using source pools in your project, you can share common nodes or project options. A source pool can share a Style Sheets between targets, or invoke a tool at a specific location in your build process.

The Project Manager supports incremental search, so you can find a specific node by typing the first part of a node's name.

Node information

Project View options let you choose what information is displayed next to each node. You can display all or a subset of these choices: build translator, code size, data size, description, location, name, number of lines, node type, Style Sheet, output, run-time nodes, project node.

Build Attributes indicate how a Target node will be built by the compiler. The Project Manager uses special glyphs in the left margin to indicate the build attributes of Target node. To apply build attributes to a node (and for a reference on the different Project Manager glyphs), right-click the node, then choose Edit Local Options from the SpeedMenu and select the Build Attributes topic.

Option settings and tools

Options set on a parent node are inherited by their child node(s). You can override settings by setting locally options on specific nodes.

Along with Style Sheets and local option overrides, you can associate tools with nodes. Tools might be standalone programs (like GREP or an alternate editor), viewers, or translators. It can also be useful to add Source Pools that serve to activate a tool during the build process.

Creating multiple-target projects

See Also

Creating multiple-target projects is very similar to creating projects with one target:

1. Create a project using File|New|Project
2. Choose Project|New Target to add a second target to your application.
3. Type a name for the second target and choose a target type (standard is the default). Choose OK to have the Project Manager add the new target to your project.

The file MULTITRG.IDE in the EXAMPLES\IDE\MULTITRG directory contains a sample project with two targets. This project file builds two versions of the WHELLO program-a 16-bit version and a 32-bit version. The project contains more information on how to use two or more targets in a project file.

With more than one target in a project, you can choose to build a single target, multiple targets, or the whole project.

Node types and colors

The Project Manager uses the following types of nodes to distinguish the different files in your project:

Node Type	Description
Project	Located at the top of the project tree, the Project node represents the entire project. All the files used to build that project are under the project node (similar to a symbolic target in a makefile). By default, the project node is not displayed in the project tree. Since it can sometimes be helpful to see the "total" picture, display the project node with Options|Environment|Project View by checking Show Project Node.
Target	A Target node represents a file that is created when its dependent nodes are built. A target can be one of a variety of target types, but is usually an .EXE, .DLL, or .LIB file that you are creating from source code. A project can contain many target nodes. For example, in a single project, you might build an executable file and two separate .DLL files, making three targets.
Source	A Source node represents a file used to build a target. Files such as .C, .CPP, and .RC are files that can be associated with source nodes.
Run-time	A Run-time node represents a file that is used during the linking stage of your project, such as startup code and .LIB files. The Project Manager adds different Run-time nodes depending on the options you specify in TargetExpert. By default, the Project Manager does not display Run-time nodes in the project tree. To display them, choose Options|Environment|Project View, then check Show Runtime Nodes.
Autodepend	Autodependency nodes are the files that your program automatically references, such as header files included in your .CPP files. By viewing autodependency nodes, you can see the files that Source nodes are dependent upon. Viewing these nodes in the Project Manager lets you easily navigate to these files (just double-click the node). By default, the Project Manager does not display Autodependency nodes; you must choose Options|Project|Make, then check Autodependencies: Cache & Display (note that you must build the project before the Project Manager can display autodependency information).

Node colors

The project manager uses the following color scheme for its nodes:

- Blue nodes represent those that were added by the programmer.
- White nodes indicate project targets.
- Yellow nodes are those that were added programmatically by the compiler (when it posts dependencies and autodependencies), by AppExpert or ClassExpert (when they add .CPP nodes), or by TargetExpert (when it adds nodes based on the target type).

The project manager uses the following glyphs:

- The local override glyph, a checkmark, is displayed when an override is set.
- The out of date glyph, a clock with an 'x' on it, is displayed when a node is out of date and will be built in the next make.

Node attributes

See also

Edit Node Attributes on the Project Tree window SpeedMenu displays the Node Attributes dialog box. These are the Node Attributes options:

Name

Enter a name for the node.

Description

Enter a comment that describes the node.

Style sheet

Select a Style Sheet for the node. Style Sheets are predefined sets of options that can be associated with a node. If <<None>> is specified, the Project Manager uses the parent options, plus any local overrides set on nodes higher in the project tree hierarchy.

- If you need to create or edit an existing Style Sheet, click the Styles button to access the Style Sheets dialog box.

Translator

Select a translator to associate with the node. A translator is any program that changes (translates) one file type to another. For example, TLINK is a translator that uses .OBJ files to produce an .EXE file.

Node type

Some node-types are assigned by the system (such as [AppExpert] and [SourcePool] nodes), but most node-types are a reflection of a file extension: [.cpp], [.exe], [.def], and so on. Source node types have default translators.

Change the node type by selecting from the drop down list of available types. Source pool, .EXE, and .DLL nodes can not be changed. You can change which viewers and translators are available through Options|Tools in the 'Advanced' dialog (see 'Applies to', 'Translate from' and 'to' fields).

- You can also edit an individual node name and type through the Project Manager's SpeedMenu Edit Node Attributes command.

Exclude debug information

Choose Exclude Debug Information to prevent debug information from being generated for the selected node.

Overlay this module

Check Overlay This Module (available only if you have chosen DOS Overlay as the Platform in TargetExpert) to have the appropriate code generated so that the module can be overlaid.

Styles button

Choose Styles to access the Style Sheet dialog box, where you can create, view or customize Style Sheets.

Tools button

Choose Tools to access the Tool Options dialog box, where you can view and customize tools for use as node translators/viewers.

Copying nodes

See also

You can copy nodes in the Project Manager by value or by reference. If you copy nodes by value, the Project Manager makes an identical, but separate, copy of the nodes. The nodes you copy inherit all the attributes from the original node, and you have the ability to modify any of the copied node's attributes.

When you copy nodes by reference, you are creating a pointer to those nodes from another location in the Project Manager. A reference node is not distinct from the original. For that reason, reference copies of nodes appear in a lighter font in the Project Manager.

To copy project nodes,

1. Select a group of nodes you want to copy.
2. Press the Ctrl key and drag the selected nodes to the new location to copy by value.

   Or

   Press the Alt key and drag the selected nodes to the new location to copy by reference.

Reference copy details

In reference copies, node attributes and local options are initially the same as they are for the original. You can then edit node attributes or local options in the reference copy without affecting the original. If you edit the original, all reference copies are updated.

- If you delete an original node, all reference copies of that node are deleted. You cannot undo such a deletion.

Dependency nodes and reference copies

Whether you make a reference copy of a source pool or normal node, the referenced copy is a mirror of the original; it contains the same set of project files as the original. While you can make file modifications to the original, you cannot add or delete files from the reference copy. Modifications to the original are echoed down through all reference copies.

Action	Original	Reference Copy
Deleting	If a node is deleted from the original, it is also deleted from the reference copy.	Dependency nodes can be deleted from the reference copy. Deleting a node from a reference copy does not affect the original.
Adding	If a nodes is added to the original, it is added to the reference copy.	You can not add a node to a reference copy.
Moving	If a node is moved out of the original, it is deleted from the reference copy.	An entire reference copy can be moved by moving the top-most node. A dependency node in a reference copy can not be moved.

Adding nodes

Choose Add Node on the Project Tree window SpeedMenu to add a node below the currently selected node. You can also add a node with the INSERT key.

The Project Manager supports drag and drop. Just select files from your File Manager, drag and drop them into place.

- New projects are created with a default project skeleton. To change the default project skeleton for a specific project type, use the Advanced Options button on the Project|New Project dialog box.

Deleting nodes

Select the node and press Del or choose Delete Node from the SpeedMenu.

- Use care when deleting nodes; you cannot undo the deletion.
- If you delete an original node, all reference copies of that node are also deleted. You cannot undo this deletion.

Deleting targets

See also

To Delete a Target

1. Select the target and open the SpeedMenu.
2. Choose Delete node.
3. The Project Manager asks if you're sure you want to delete the target. Choose OK to delete the Target node and all it's dependencies.

- Use care when deleting target nodes; you cannot undo the deletion.

Grouping files with Source Pools

See also

What is a Source Pool?

A *Source Pool* is a collection of nodes that a Target node references as a single unit. When a Target node references a Source Pool, the nodes in the Source Pool take on the options and target attributes set in the Target node.

Because you can reference a single Source Pool from different Target nodes, you can easily manage files that different Target nodes share. For example, you can create both 16- and 32-bit targets from the same set of source files by referencing one Source Pool from both targets; the Project Manager compiles the targets according to the settings made in the Target nodes. You can then update the single Source Pool and modify all the Target nodes that reference the same set of files.

You can also create a Source Pool that represents a specific option set or that contains a customized tool.

Using Source Pools

Among other things, you can use Source Pools to

- Share file sets
- Share option settings (with Style Sheets)
- Call tools at specific build locations

Sharing file sets

Source Pools are particularly helpful when you have a set of header files that needs to be included throughout your project. If you place the header files in a Source Pool, you can easily reference them by each node that uses them. Then, if you need to make changes to the group of header files, you only have to update the one Source Pool. Sharing a set of source files is also invaluable when you are building two versions of a library—using a Source Pool insures that both library versions are always in-sync.

Sharing option settings

Source Pools are also useful when you want multiple Target nodes to share a single set of options settings. In this case, the settings are contained in a single Style Sheet. For example, if three targets in a project use the same Style Sheet, each Target node can reference a Source Pool that contains that Style Sheet. If you then need to update the Style Sheet (for example, if you want to change from compiling with debug information to compiling without it), you can update all the targets by modifying the single Style Sheet.

For flexibility, break up option settings into small easy to handle Style Sheets (e.g., one for compiler options, one for librarian options, one for linker options, one for debug information, and so on). Attach Source Pools to these smaller Style Sheets, and you can pop settings in and out of your targets by commenting in or out the appropriate Source Pool.

Calling tools

You can also use Source Pools to activate a customized tool at a specific point in the build process. For example, if a Source node requires a custom translator, you can reference a Source Pool that contains the tool from that Source node. Customized tools, for example, can create open a DOS shell and use transfer macros (e.g., remove debug information, copy files to a delivery directory, link in libraries, point to where header files can be found, or create import libraries). For more information on using custom tools in Source Pools, see the sample project DELIVER.IDE in the EXAMPLES\IDE directory.

Creating Source Pools

When you create a Source Pool, you create a Target node with a group of nodes under it. The Target node cannot be compiled—to compile the nodes in a Source Pool, you must copy the Source Pool to a Target node. Source Pools work to your best advantage when you copy them by reference.

To create a Source Pool

1. In the Project Manager, position the insertion point where you want to insert the Source Pool.
2. Choose Project|New Target.
3. Enter the name for the Source Pool.
4. Select Source Pool from the Type list and choose OK to create a Source Pool target node in your project.
5. Select the new Source Pool, then press INS to open the Add To Project List dialog box.
6. Select the source files you want, then choose OK to add them to the Source Pool.
7. Copy the Source Pool by reference by holding down the Alt key and dragging the Source Pool to the desired target nodes.

Open the sample project SRCPOOL.IDE in the EXAMPLES\IDE directory. This project includes a text file that describes how the project incorporates a Source Pool.

▪ The Project Manager marks a Source Pool by placing an X to the left of node in the Project Manager. Because you cannot build Source Pools directly, they are marked "exclude from parent."

Project Manager commands

Project Manager

Below is a list of common tasks you can perform in the Project Manager. Navigating in the Project Manager contains a reference to moving through the project tree.

Editing:

- Add a Node
- Change the Project View
- Collapse a Node
- Collapse the Project
- Copy a Node
- Default Action for Node
- Delete a Node
- Demote a Node
- Expand a Node
- Expand the Project
- Find a Node
- Move Cursor to Bottom
- Move Cursor to Top
- Move Node(s) Up or Down
- Open the SpeedMenu
- Page Through the Project
- Promote a Node
- Reference Copy a Node
- Scroll Right or Left
- Search for a Node
- Select Contiguous Nodes
- Select Non-Contiguous Nodes
- Select One Node

Management Tasks:

- Add Targets
- Build Part of a Project
- Control Build Attributes for Nodes
- Convert Old Projects
- Create A Multiple Target Project
- Set Local Options
- Set Node Attributes
- Share Style Sheets
- Use Reference Copies of Nodes
- Use Source Pools
- Use Style Sheets
- Use the TargetExpert to Create Target Types
- View Option Settings

Select one node

- Click the node
- Type node name and the selection bar moves incrementally to nearest match
  ESC key exits incremental search mode
- Up/Down Arrow move selection bar up or down (does not wrap)

Find (search for) a node

- Left Click on a node
- Type node name and selection bar moves incrementally to nearest match; Ctrl+S finds next match
  ESC key exits incremental search mode
- Up/Down Arrow move selection bar up or down (does not wrap)

Select contiguous nodes

First, select a node (Left click, type name, or use Up/Down Arrow). Then,

- Shift+Left click to select nodes between first node and current mouse cursor
- Shift+Up/Down arrow to select additional nodes

Select non-contiguous nodes

1. Select a node (Left click, type name, or use Up/Down Arrow)
2. Ctrl+Left click each additional node

Move node(s) up or down

First, select a node(s). Then,

- Left click drag to move node(s). Nodes moved with the mouse become a child of the drop target
- Alt+Up/Down arrow move node(s) one line up or down
- Non-contiguous nodes become contiguous nodes when moved.

Promote a node

1. Select a node
2. Alt+Left arrow moves node up through levels of dependencies

Demote a node

1. Select a node
2. Alt+Right arrow moves node down through levels of dependencies

- You can also demote a node by moving it with the mouse. Nodes moved with the mouse become a child of the drop target.

Expand a node

- Left click on folder icon marked + to expand node
- Plus expands a node marked +
- Spacebar expands a node marked +

Collapse a node

- Left click on folder icon marked - to collapse node
- Minus collapses a node marked -
- Spacebar collapses a node marked -

Add a Node

- Insert adds a node below the selected node
- Select Add Node on the SpeedMenu

Choose the files you want associated with nodes. If the files you type do not exist in the current directory, the IDE creates them.

- Do not use Insert to add target nodes. To add a target node, use Project|New Target. This opens the New Target dialog box, where you can select target type, platform, model, and libraries.

Delete a node

- Delete deletes the selected node
- Select Delete Node on the SpeedMenu
- Use care when deleting target nodes. You cannot undo the deletion.
- Use care when deleting a node that has been reference-copied. All reference copies of that node are also deleted. You cannot undo the deletion.

Change the Project view

Use Options|Environment|Project View to set options that specify what types of nodes appear in the Project Manager and what information appears beside each node.

The options to choose from are Build Translator, Code Size, Data Size, Description, Location, Name, Number of Lines, Node Type, Style Sheet, Output, Show Runtime Nodes, and Show Project Node.

Expand the project tree

- The * key expands project tree.

Collapse the project tree

- The - key collapses project tree.

Scroll project tree left or right

- Left/Right Arrow scrolls left or right in the project tree.
- Scroll Bar scrolls left or right in the project tree.

Move cursor to top of project tree

- Home moves the cursor to the top of the project tree.
- Scroll Bar moves you up or down through the project tree.

Move cursor to bottom of project tree

- End moves the cursor to the top of the project tree.
- Scroll Bar moves you up or down through the project tree.

Page through the project tree

- Page Up/Page Down pages up or down through the project tree.
- Scroll Bar moves you up or down through the project tree.

Copy a node

- Ctrl+Left Click Drag to make a copy of the selected node and make it a child of the drop target.

This creates an identical, but separate copy of the node in the location you specify. Nodes that have been copied inherit all attributes from the original node and you can modify attributes for either the original or the copy.

Reference copy a node

- Alt+Left click drag to make a reference copy of the selected node and make it a child of the drop target. Reference copies appear in a lighter font to identify them in the project tree.

A reference copy acts as a pointer to a node located elsewhere in the project. A reference copy is not a unique node. If you modify the original node (attributes or dependents), the reference copy is also modified. If you delete an original node, all reference copies of that node are also deleted. You cannot undo this deletion.

Open SpeedMenu

- Right Click opens the SpeedMenu for the node at the mouse cursor
- Alt+F10 opens the SpeedMenu for the selected node

Default action for node

Double-click a node to perform the node’s default action. You can find the default action by right-clicking the node and choosing View. The first "viewer" listed is the default action for the selected node.

Navigating in the Project Manager

See also

The Project Manager tree can be traversed with the mouse or the keyboard.

- The Project Manager supports *incremental searching*, so you can quickly find a node by typing the node name. Incremental searching finds the first node in the Project Manager that matches the letters you type. Press Ctrl+S to find the next match.

Task	Keyboard	Mouse
Add Node	Insert	Right Click|Add Node
Collapse hierarchy	Minus	Click parent node
Collapse/Expand node	Spacebar	
Copy Node		Ctrl+Left Click Drag
Default action for node	Enter	Double Click
Delete Node	Delete	
Demote a node	Alt+RightArrow	Left Click Drag
End node search	Esc	
Expand hierarchy	+ (Plus)	Click parent node
Expand entire hierarchy	* (asterisk)	
Find a node	Incremental search (start typing)	
Move down in project	DownArrow	Scroll Bar
Move node down	Alt+DownArrow	Left Click Drag
Move node up	Alt+UpArrow	Left Click Drag
Move to bottom of hierarchy	End	Scroll Bar
Move to top of hierarchy	Home	Scroll Bar
Move up in project	UpArrow	Scroll Bar
Open SpeedMenu	Alt+F10	Right Click
Page down	PgDn	Scroll Bar
Page up	PgUp	Scroll Bar
Promote a node	Alt+LeftArrow	
Reference Copy Node		Alt+Left Click Drag
Scroll left	LeftArrow	Scroll Bar
Scroll right	RightArrow	Scroll Bar
Select a nodeNodesDef*	Up/DownArrow	Left Click
Select Contiguous nodes	Shift UpArrow	Shift Left Click
Select Non-Contiguous nodes		Ctrl Left Click

Converting old projects

See also

The Project Manager can load and use projects from previous versions of Borland C++. Choose Project|Open Project, then type the name of the old project file. You can also change the search attributes from *.IDE to *.PRJ to list the old 3.0 and 3.1 projects.

The Project Manager converts the old project to a new one. Be sure to save the new project if you want to keep using it with this version of Borland C++. To save the project, choose Options|Save. Make sure Project is checked, then click OK. The new project is saved with the old name and the new .IDE extension.

Interoperability with Borland C++Builder

See also

Porting Borland C++Builder .MAK files to Borland C++

You can add Borland C++Builder makefiles as nodes to your Borland C++ projects. The IDE automatically creates an .IDE file with the correct source nodes.

To add a Borland C++Builder makefile to your project:

1. Select a node in the Project Manager and right-click.
2. Choose Add Node. The Add to Project List dialog box is displayed. In the Files of Type list, choose Makefiles (*.MAK).
3. Select the name of the makefile to add.
4. Click OK or Open.

- Once you include a Borland C++Builder node in your project, you will not be able to use TargetExpert to change target information. Any changes you make to the Borland C++Builder target (such as adding source modules) will not be written back to the makefile.
- You can also use File|Open to load in Borland C++Builder MAK files.

Porting Borland C++ projects to C++Builder

You can use `IDETOMAK.EXE` to convert your .IDE file to a Borland C++Builder .MAK file. `IDETOMAK` is in the `BC5\BIN` directory.

Compiling (Project Tree Window SpeedMenu)

When you select Compile on the Project Tree window SpeedMenu, the Project Manager translates the selected nodes (not the project itself).

- Additional ways to compile in the Project Manager are to double click on the node or select Compile from the Project Menu.

A Compile Status information box displays the compilation progress and results. Choose OK when compilation is complete.

If any errors occurred, the Message window becomes active and displays and highlights the first error or warning.

Building part of a project

See also

You can build part of a project three ways:

- Select a node and use Build node from the SpeedMenu to build it and all of its dependents.
- Use Make node from the SpeedMenu, to build only the out-of-date nodes.
- Translate a single node.

Build a node and its dependents

1. Choose the node you want to build.
2. Right-click the node and choose Build node from the SpeedMenu.

All the dependent nodes are built regardless of whether they're out-of-date.

Build a project using make node

1. Choose the node you want to build.
2. Right-click the node and choose Make node from the SpeedMenu.

Make node builds only the nodes that are not current.

Translate the individual node

Suppose you're working on a file and you just want to see if it will compile (you don't want to compile the entire project).

1. Select the node you want to translate.
2. Either
   a. Choose Project|Compile from the main menu.
   b. Open the SpeedMenu and choose a translation command. Default translation commands are provided for many nodes. For example, if you've selected a .CPP file, the SpeedMenu for that node contains a C++ Compile command, which compiles only the selected .CPP node.
   c. Use your own translator. Add translators to the Tools Menu or to the SpeedMenu with the Options|Tools advanced editing feature.

- Project|Compile translates the current node if the Project Manager is selected. If an Edit window is selected, Project|Compile translates the text in the editor.

Make node (Project Tree Window SpeedMenu)

Choose Make Node on the Project Tree window SpeedMenu to MAKE all targets. Starting at the bottom of the project tree, the Project Manager checks source file dates and times to see if they have been updated since the corresponding Target node was generated. If so, the Project Manager rebuilds those targets. It then moves up the project tree and checks the times and dates of those targets and sources. In this fashion, the Project Manager checks all the nodes in a project and builds all of the out-of-date files.

If no project is loaded, Make Node causes BCW to generate an .EXE file whose name is derived from the name of the file in the Edit window.

- The Make Node command rebuilds only the files that are not current.

Build node (Project Tree Window SpeedMenu)

The Build Node command on the Project Tree window SpeedMenu rebuilds all the files in the current project, regardless of whether they are out of date.

This option is similar to Make Node on the SpeedMenu, except that Build Node rebuilds all the files in the project, regardless of the file dates.

Build Node builds the project using the Default Project Options Style Sheet unless you have attached a different Style Sheet to a node or overridden the options locally.

For example, if you have a project with an .EXE target that is dependent on a .CPP file, and the .CPP is dependent on two .H files, the Project Manager checks the dependencies for the two .H files first, then it compiles the .CPP file to an .OBJ, and finally, the project uses the new .OBJ file to link the .EXE.

Build Node:

1. Deletes the appropriate precompiled header (.CSM) file, if it exists.
2. Deletes any cached autodependency information in the project.
3. Builds the node / executes the translator.

If you abort a Build Node command by pressing ESC or choosing Cancel, or if you get errors that stop the build, you must explicitly select the nodes to be rebuilt.

Local options

See also

Inherited options or Style Sheet options can be overridden at the node level. Set options for an individual node by selecting Edit Local Options on the Project Tree window SpeedMenu or by selecting Edit Local Options from the Edit Local Options on the Edit window SpeedMenu when no project is loaded. The local options dialog box displays where the node is located in the Project Manager and allows you to set options for that node.

Once options have been set, they become local overrides associated with the node. Local Override are useful when you use a Style Sheet (perhaps inherited from a parent node).

To set local options

1. Select (highlight) a node for which you want to override options.
2. Choose Edit Local Options from the SpeedMenu. The Project Options dialog box is displayed.
3. Select the option you want to override. Selecting an option automatically activates the Local Override checkbox. The Local Override checkbox is grayed if no option is selected. You can tab over to an option if you want to see if the Local Override box is checked, but not change the setting for that option. You must select each individual option to see if it is overridden. If you find yourself overriding more than one or two options, you might want to create a separate Style Sheet instead of using Local Override. Any settings you make take place immediately.

- If you are modifying a stylesheet, the Local Override checkbox is replaced by the Include checkbox which it performs the same function, except that it applies to the active style sheet instead of a node in the Project Manager.

Note: When an override is set, a checkmark is displayed next to the node.

To undo an override

1. Select (highlight) a node for which you want to override options.
2. Choose Edit local options from the SpeedMenu. The Project Options dialog box is displayed.
3. Select the option and clear the Local Override or Include checkbox.

Setting options locally

A Local Options dialog box contains the same options as the Options|Project dialog box, except that changes you make here affect only a selected node. This is also where Build Attributes for an individual node are set.

A Local Options dialog box displays when you:

1. Choose Edit Local Options from the Project Tree window SpeedMenu.
2. Choose View Options Hierarchy from the Edit window SpeedMenu when no project is loaded.
3. Select a node, choose View Options Hierarchy from the Project Tree window SpeedMenu, then double-click in the Components list on the option you want to override.

Overriding an option

Select the option you want to override. Selecting an option automatically activates the Local Override box. The Local Override box is grayed if no option is selected. You can tab over to an option if you want to see if the Local Override box is checked, but not change the setting for that option. You must select each individual option to see if it is overridden. If you find yourself overriding more than one or two options, you might want to create a separate Style Sheet instead of using Local Override. Any settings you make take place immediately.

Button	Description
OK	Accepts changes to all options and return to the Project Manager.
Undo Page	Restores the current screen of options to the settings in effect when you selected the Options Settings dialog box.
Cancel	Restores all options to the settings in effect when you opened the Options Settings dialog box. (Same as close box.)
Help	Gets a summarized description of the current screen of options. For detailed Help on an individual option, select (highlight) an option and press F1.

To undo an override

To undo an override, select the option and uncheck the Local Override box.

To override multiple options

Options dialog boxes use a dual window format. A list of topics is displayed in the left window.

Topics in the left window expand to display sub-topics. Select a sub-topic and the options associated with that sub-topic are display in the right window.

When you first open an Options dialog box, the list of topics is collapsed. This is indicated with a + sign beside any topic which can be expanded. Highlight a topic to display a brief summary of the types of options available under that topic.

To expand or collapse a topic, double-click on it or highlight it and press Enter or use the commands on the Options dialog box SpeedMenu. When expanded, the + changes to a -, and a list of sub-topics is displayed under the topic.

To override multiple options, choose each sub-topic and options display on the right. You can change settings on one or several pages, without leaving the Options dialog box. Options are initially set to their default values.

View options hierarchy (Project Tree Window SpeedMenu)

See also

Choose the View Options Hierarchy command on the Project Tree window SpeedMenu to display the Options Hierarchy dialog box where you browse through the option settings for your project, node by node.

Project Options

A node tree on the left offers a graphical representation of your project. Browse through the tree and select a node to view or change the option settings.

Outliner Symbols

- **+** indicates a collapsed node or branch (sublevel categories not displayed)
- **-** indicates a fully expanded node or branch

Options list

The Options list shows each node in square brackets, followed by the name of that node's Style Sheet. It also lists any options that are overridden for that node. View Options Hierarchy lets you see what options are sent to dependent nodes.

To change an option for the project at a global level, highlight an option within the Default Project Options section and double-click. You can also change global options by double clicking the Project node.

- By default, the project node is not displayed in the Project Manager. To display the project node, choose Show Project Node in the Options|Environment|Project View.

Edit button

Press the Edit button to edit option settings for the currently selected node. Local Options are displayed. You can also double-click on a specific option to display the Local Options page that contains that option, where you can make changes for the selected node.

Edit button (Options Hierarchy dialog box)

Press the Edit button to edit option settings for the currently selected node. Local Options are displayed. You can also double-click on a specific option to display the Local Options page that contains that option.

If the option belongs to a Style Sheet, you are editing the Style Sheet. If the option is overridden at the node level, you are editing the Local Options for that selected node.

Style sheets

See also

A Style Sheet is a group of option settings. Option settings control how target nodes in your project are built. You can attach Style Sheets to entire projects or to individual nodes in a project. You can attach one or more Style Sheets to your entire project or assign one or more Style Sheets to individual nodes in your project.

When you initially create a project, it inherits the settings of the default project when you choose Project|New Project. The new project is given the settings of the Default Style Sheet. If some components in your project require different settings, you can attach different Style Sheets to those nodes or use Local Options to override the default project Style Sheet settings.

- The sample project STYLESHT.IDE in the directory \EXAMPLES\IDE\STYLESHT illustrates the use of Style Sheets and contains a text file that explains the use of Style Sheets. This file uses Style Sheets for each of the targets (two versions of WHELLO).

The available style sheets list

Choose Options|Style Sheets on the main menu (or click the Styles button on the Edit Node Attributes dialog box) to access the Style Sheets dialog box. where you create, compose, copy, edit, rename, or delete from the list of Style Sheets that are available for your project.

The list of available Style Sheets is associated just with the current project. If you create a Style Sheet for one project, then open a second project, the new Style Sheet will not be included in the second project's list of available Style Sheets. However, Sharing Style Sheets between projects is possible, allowing styles and tools from the current project to be inherited by new projects.

At build time

When a node is built:

- The Project Manager uses the node's Style Sheet and any Local Overrides
- If a node does not have a Style Sheet, the Project Manager uses the Style Sheet of the parent.
- If the parent node does not use a Style Sheet, the Project Manager looks at the next parent, continuing up the hierarchy until it reaches the project node.

Changing default project options

To modify default project options, choose Options|Project. This displays the Project Options dialog box, where you set options for your entire project.

You can also change default settings by choosing Options|Style Sheets on the main menu (or click the Styles button on the Edit Node Attributes dialog box). Select the Default Project Options Style Sheet from the list of available Style Sheets.

Attaching style sheets to nodes

Sometimes different nodes in a project need to be built with option settings that are different than those in the project Style Sheet. For example, you might want to compile .C files with one set of options but .CPP files with another. Or, you might want to build one target with 16-bit options and another with 32-bit options.

To attach an existing Style Sheet to a node:

1. Select the node and right-click.
2. Choose Edit node attributes.
3. Select a Style Sheet from the combo box.
4. Click OK.

- To display the name of any attached Style Sheet next to nodes in the Project Manager, choose Options|Environment Project View and select Style Sheet.

Sharing style sheets

See also

There are two ways to share Style Sheet between projects:

- inheriting style sheets from another project
- editing the .PDL file associated with a project

Inheriting style sheet settings

When you create a custom Style Sheet, that Style Sheet remains with the project for which it was created; it doesn't get added to the list of predefined Style Sheets. However, if you want a new project to use one of your custom Style Sheets, you can do so by letting a new project inherit settings from another project.

To pass Style Sheets (and user defined tools) from an existing project to a new project, edit your BCW5.INI file to make sure it includes the following lines:

```
[Project]
inherit=1
```

This allows Style Sheets to be inherited from an open project.

Next, open a project that contains the settings you want to pass to a new project. Choose Project|New Project to create the new project. At creation, the new project inherits the option settings and user defined tools of the project that was open.

- If you set the BCW5.INI `inherit` setting to `0`, your new project will inherit settings from the Default Project Settings. Setting `inherit` to `2` will cause new projects to inherit the factory default settings.

Editing the project description language files

You can also share Style Sheets across projects by editing the Project Description Language files (.PDL) associated with your projects.

- Be careful if you choose to edit .PDL files. If a .PDL file is corrupted, the Project Manager will not be able to read it. You may want to make a backup copy of the .PDL file before you begin making changes.

A .PDL file is a text file that contains information about the Style Sheets and Tools used in your project. Style Sheet and Tools information can be copied from one .PDL file to another, allowing you to quickly modify Style Sheet and tools for your projects.

- .PDL files are only generated when `saveastext=1` and `readastext=1` entries are present in the `[Project]` section of your BCW5.INI file. The `saveastext` setting tells the IDE to save a .PDL file whenever a project is saved. The `readastext` settings tells the IDE to update an .IDE file if its associated .PDL file is newer that the .IDE file.

To edit the .PDL file:

1. Open the .PDL file containing the Style Sheet you want to share. You can open the .PDL file using any text editor.
2. Search for Subsystem = Style Sheet. Then, scan for the desired Style Sheet by name. For example, if you created a Style Sheet called MYSTYLE, you'll see a section in the .PDL file that starts `{ StyleSheet = "MYSTYLE"`.
3. Copy all the text from the beginning to the ending brace. You can copy more than one Style Sheet.

   To share a user-defined tool, copy the section that reads `Subsystem=<tool>`.
4. Open the .PDL file that is going to use the copied Style Sheet.
5. Find the section for Style Sheets, then paste the copied text to the end of the existing Style Sheet list.
6. Save the .PDL file that received the copied Style Sheet. When you open a project file, the Project Manager checks the date of the .PDL file with the same name as the .IDE file. If the .PDL file is newer than the .IDE file, the contents of the .PDL file is added to the .IDE file.

After transferring Style Sheets, it is a good idea to reset the `saveastext` and the `readastext` settings

in the BCW5.INI file to `0`. This tells the IDE to not save or update the .PDL files.

Nodes

The project tree uses nodes to visually represent the files and components contained in your project. Among other things, nodes can represent source files, target files, translators, viewers, and source pools.

Translator

A translator can be any program that changes (translates) one file type to another. For example, the compiler is a translator that uses .C and .CPP files to create .OBJ files. TLINK is a translator that uses .OBJ files to produce an .EXE file.

Viewer

A viewer is a tool that lets you see the contents of a node. For example, an editor is a viewer that lets you examine the code in a .CPP file. On the local menu for a .CPP node, you'll see the Text Edit command. The default editor for the Text Edit view is the IDE editor.

Other node types have other viewers available. For example, Resource Workshop can view .RC files. You can't view an .EXE node in a text editor, but you can choose to view it using the integrated debugger, Turbo Debugger for Windows, the Browser, or even as an executing program.

Style Sheets

A Style Sheet is a group of option settings. Option settings control how targets in your project are built. You can attach one or more Style Sheets to your entire project or assign one or more Style Sheets to individual nodes in your project.

Target nodes
A file created when its dependent nodes are built. A target is usually an .EXE or .DLL that you are creating from source code. A project can contain many target nodes. For example, in a single project, you might build an executable file and two separate .DLL files, making three targets.

Source pools

Source Pools are a collection of nodes in the project that are typically marked "exclude from parent" and not built. Use Source Pools to share common source or header files between targets (great when multi-targeting). Use Source Pools to share Style Sheets between nodes and update it in one place. You can also use Source Pools to invoke tools at strategic points in your build process.

Tools

Tools are programs and utilities that can be run without leaving the IDE. Tools can be installed on the Tool Menu, or associated with project nodes. They can be standalone programs (like GREP, Turbo Debugger, or an alternate editor). Tools can be used to build nodes in the project. They can be used to view node contents or carry out an action at a specific point in your program compilation process. You can specify program arguments when you create a tool, or opt to be prompted for them at run-time.

TargetExpert

See also Add-on Products

TargetExpert lets you specify or change the target type of a Target node in the Project Manager.

There are two versions of TargetExpert:

- The New Target version lets you add a new target node to your project tree.
- The standard TargetExpert version lets you change the settings of an existing target node. For example, TargetExpert lets you modify a target node so that the compiler generates a 32-bit Windows DLL instead of a 16-bit Windows DLL.

To change a node's target type, right-click the target node in the Project Manager, then choose TargetExpert from the SpeedMenu. Note, however, that TargetExpert does not let you change the target type of a Source Pool or of an AppExpert target.

- You can use TargetExpert to set the target type for source files that are not associated with a project. Just right-click in the Edit window when no project is loaded, then select TargetExpert from the SpeedMenu.

The TargetExpert dialog box contains the following option setting categories:

- Target type - type of target (output) file
- Platform - the desired platform for your project
- Target model - the memory model for your program
- Frameworks - the class libraries used in your program
- Controls - the types of controls displayed by your program
- Libraries - which libraries or defines to use in your application
- Library versions - specify the type of library framework you want to work with
- Math support (DOS targets only) - specify floating-point-support

New Target (TargetExpert)

See also

The New Target version of the TargetExpert dialog box displays whenever you create a new project or when you add a new target node to an existing project.

To add a new target node to an existing project:

1. Choose Project|New Target.
2. Enter a name and target type in the Add Target dialog box.

   The New Target dialog box is then displayed.
3. Fill in the appropriate options.

The options in the New Target dialog box are the same as those in TargetExpert with the addition of the following options:

- Project Path and Name - the directory, path, and name for your project
- Target Name - the name of the target that you are adding to your project
- Advanced Options - opens the Advanced Options dialog box, which lets you specify the default nodes generated with a new target
- You can also use the Browse button to select a target file.

Project path and name

Enter the path and an eight-character name for your project. You can also use the Browse button to select a directory and path to the project file.

- If you specify a directory that does not exist, Borland C++ creates the directory.

Target name
Type the name of the new project target. This is usually the name of the .EXE or .DLL that you want to create. You can also use the Browse button to select a target file.

Advanced Options button

The Advanced Options button on the Target Expert dialog box displays the Advanced Options dialog box.

The Project Manager creates new targets with a default project skeleton that usually consists of a .CPP node, an .RC node, and a .DEF node. The Advanced Options dialog box lets you change the default project skeleton for your new target.

Browse button

The Browse button opens a dialog box where you can search for path or file-name information.

Advanced Options dialog box

See also

When you create a new Target node, the Project Manager generates a skeleton project tree for the target. The project tree represents the files and components that it uses to generate the Target node. The nodes added to the project tree depend upon the type of the Target you specify in TargetExpert. You can change these default settings using the Advanced Options dialog box.

Extension	File Type
.cpp node	Creates a C++ language Source node.
.c node	Creates C language Source node.
No source node	Creates a Target node that doesn't use a source node. Use this option when you want to create a Source node that uses a different name than the Target node (you must then add the Source node manually).
For Windows programs	
.rc	Creates a Source node that is associated with a resource script.
.def	Creates a Source node that is associated with a Windows module definition file, which is used by the linker.

Default project tree nodes

When you change the settings in the Advanced Options dialog box, those settings become the default settings for that specific target type. Subsequent new projects of that target type are created with a skeleton project tree containing the newly defined default set of project nodes.

The system default target options for the Application, Dynamic Library, and EasyWin Target types are:

Application type	Platform	Default source nodes generated
Application (.exe)	Win16 or Win32	.CPP, .RC, and .DEF
Dynamic Library (.dll)	Win16 or Win32	.CPP, .RC, and .DEF
EasyWin (.exe)	Win16	.CPP, .RC, and .DEF
Application (.exe)	DOS Standard	.CPP
Application (.exe)	DOS Overlay	.CPP

- The Project Manager also adds the proper libraries to the project tree, depending on the settings you choose in TargetExpert.

Target types

NewTarget

Select the type of target (output) file you want to generate. The target type you select affects the resulting project skeleton that is created for your project. The selected node's extension changes to reflect the target type selected.

Choose from the following:

Target Type	Explanation
Application [.exe]	Creates a standard application (.EXE)
Dynamic Library [.dll]	Creates a dynamic library (.DLL) file
EasyWin [.exe]	Creates a character-mode application that runs under Windows
Static Library (for .exe) [.lib]	Creates a static library (.LIB) file for an .EXE
Static Library (for .dll) [.lib]	Creates a static library (.LIB) file for a .DLL
Import Library [.lib]	Creates a import library (.LIB) file
Windows Help [.hpj]	Creates a Windows Help project (.HPJ) file

Application (.exe)

Application (.exe) - generates an Application project and creates the current default project skeleton for that target type.

Dynamic library (.dll)

Dynamic Library (.dll) - generates a Dynamic Library project and creates the current default project skeleton for that target type.

EasyWin (.exe)

EasyWin (.exe) - generates an .EasyWin project and creates the current default project skeleton for that target type. An EasyWin project is a character-mode (DOS) application that runs under Windows.

Static library (.lib)

Static Library (for .exe) [.lib] - generates a Static Library file for use in an executable.

Static Library (for .dll) [.lib] - generates a Static Library file for use in a dynamic link library.

Import library (.lib)

Import Library (.lib) - generates an Import Library project.

Windows help (.hpj)

Windows Help (.hpj) - generates a Windows Help file that is usually accessed from a Windows application (.exe).

Project platform

See also

Select the desired platform for the target node.

Windows 3.x (16)
Targets 16-bit Windows platforms (Windows 3.*x*).

Win32
Targets 32-bit Windows platforms (Windows NT and Windows 95).

DOS Standard
Targets DOS applications not designed to be safe for overlays.

DOS Overlay
Targets DOS applications that are overlay-safe.

Exception handling and DOS overlays

NewTarget

If you have a DOS C++ overlay program that contains exception-handling constructs, there are a number of situations that you must avoid. The following program elements cannot contain an exception-handling construct:

- Inline functions that are not expanded inline
- Template functions
- Member functions of template classes

Exception-handling constructs include user-written `try` / `catch` and `_ _try/_ _except` blocks. In addition, the compiler can insert exception handlers for blocks with automatic class variables, exception specifications, and some `new` / `delete` expressions.

If you attempt to overlay any of the above exception-handling constructs, the linker identifies the function and module with the following message:

```
Error: Illegal local public in function_name in module module_name
```

When this error is caused by an inline function, you can rewrite the function so that it is not inline. If the error is caused by a template function, you can do the following:

- Remove all exception-handling constructs from the function
- Remove the function from the overlay module

You need to pay special attention when overlaying a program that uses multiple inheritance. An attempt to overlay a module that defines or uses class constructors or destructors that are required for a multiple inheritance class can cause the linker to generate the following message:

```
Error: Illegal local public in class_name:: in module module_name
```

When such a message is generated, the module identified by the linker message should not be overlaid.

The container classes (in the BIDS?.LIB) have the exception-handling mechanism turned off by default. However, the diagnostic version of BIDS throws exceptions and should not be used with overlays. By default, the string class can throw exceptions and should not be used in programs that use overlays.

Target model

NewTarget

Select the desired memory model for your 16-bit and 32-bit targets:

16-bit targets

- *Tiny* (DOS 16-bit only) sets four segments registers (CS, DS, SS, and ES) to the same starting address, giving you a total of 64K for all your code, data, and stack.
- *Small* uses different code and data segments, giving you near code and near data.
- *Medium* gives you near data and far code.
- *Compact* is the inverse of the Medium model, giving you near code and far data.
- *Large* gives you far code and far data.
- *Huge* (DOS 16-bit only) is the same as Large model, but allows more than 64K of static data.

32-bit targets

- *GUI* supports Win32 GUI applications.
- *Console* supports Win32 console applications.
- Windows and DOS have different uses for the DS and SS segments. In Windows, they always reference the same value; however, in DOS, these segments are different for the Compact, Large, and Huge memory models.

Frameworks

NewTarget

Frameworks are the class libraries upon which you build your application. Choose any combination of the following libraries for your application.

- Class library - a template-based container class library that uses encapsulation so you can maximize container storage with a minimal amount of reprogramming. When you select this option, TargetExpert links the appropriate BIDS*xxx*.LIB or BIDS*xxx*.DLL libraries to your program.
- OWL (Object Windows Library) - an object-oriented class library that encapsulates the behaviors (application-level and window-level) commonly performed in Windows applications.
- OCF - a set of classes (titled ObjectComponents) that simplifies the creation of OLE applications in C++. ObjectComponents implements a set of high-level interfaces on top of OLE.
- MFC - a set of classes (Microsoft foundation classes) that you can use to code Windows applications. Borland C++ supports MFC versions 3.*x* and 4.*x*.

MFC 3.2 compatibility

To build v3.2 Microsoft foundation class (MFC) libraries with Borland C++, you must install a copy of MFC 3.2. (MFC 3.2 is not supplied with Borland C++ 5.0.)

For full instructions on using Borland C++ to create MFC 3.2 libraries, see the file README.TXT located in the MFC32 directory under the Setup directory on your Borland C++ 5.0 CD.

MFC 4.0 compatibility

To build v4.0 Microsoft foundation class (MFC) libraries with Borland C++, you must install a copy of MFC 4.0. (MFC 4.0 is not supplied with Borland C++ 5.0.)

For full instructions on using Borland C++ to create MFC 4.0 libraries, see the file README.TXT located in the MFC40 directory under the Setup directory on your Borland C++ 5.0 CD.

Class library

See also

Borland C++ includes a template-based container class library that uses encapsulation so you can maximize container storage with a minimal amount of reprogramming. When you select this option, the appropriate BIDSxxx.LIB or BIDSxxx.DLL libraries are linked into your program.

These classes build containers from ADTs (abstract data types), and use FDS (fundamental data structure) as an underlying data structure. Together ADTs and FDSs determine a containers storage strategy.

- FDS are low-level containers you can instantiate. FDSs are the underlying data structures upon which you build containers and they do not necessarily need an accompanying ADT.
- ADTs are high-level containers which have at least one pure virtual function. The existence of a pure virtual function means that the ADT container classes cannot be instantiated without an underlying FDS.

Diagnostic version

A diagnostic version of these libraries is supplied on the CD version of Borland C++. If you are using the disk version of the product, then you must build a diagnostic version. Source is provided.

To build the Class Library diagnostic libraries, use the following make line:

```
MAKE -DDBG
```

- See the ClassLib makefile for other options.

ObjectWindows library

NewTarget

ObjectWindows is an object-oriented class library that encapsulates the behaviors (application-level and window-level) that Windows applications commonly perform. It provides an exciting way to develop Windows applications by providing:

- a consistent, intuitive, and simplified interface to Windows
- supplied behavior for window management and message-processing
- a basic framework for structuring a Windows application

You inherit this base functionality, which leaves you free to concentrate your efforts on the unique requirements of your application.

Diagnostic version

A diagnostic version of these libraries is supplied on the CD version of Borland C++. If you are using the disk version of the product, you must build a diagnostic version. Source is provided.

To build the OWL diagnostic libraries, use the following make line:

```
MAKE -DDIAGS
```

- See the ObjectWindows makefile for other options.

ObjectComponents library

NewTarget

The ObjectComponents library (OCF) contains a framework of classes for creating OLE applications in C++. ObjectComponents is designed to make linking and embedding easier.

Microsoft's OLE operating system extensions require a programmer to implement a variety of interfaces depending on the tasks an application undertakes. To simplify programming, Borland implements a smaller set of high-level interfaces on top of OLE. The simplified interfaces reside in a support library called BOCOLE.DLL. This support library provides default implementations for many standard OLE interfaces.

For More Information

To learn more about the Borland ObjectComponents library and Borland C++ support for OLE, see the ObjectComponents library Help (OCF.HLP).

Diagnostic version

A diagnostic version of these libraries is supplied on the CD version of Borland C++. If you are using the floppy-disk version of the product, you must build a diagnostic version. Source is provided.

To build the OCF diagnostic libraries, use the following make line:

```
MAKE -DDIAGS
```

Controls

NewTarget

The Controls options let you choose what type of controls your application uses. Borland C++ supports the following types of controls:

- BWCC - lets you use Borland's graphical Windows controls in your application.
- VBX - lets you use VBX (Visual BASIC) controls in your application.
- CTL3D lets you use the standard Windows controls in your application.

BWCC controls

The Borland Windows Custom Control Library (BWCC) is a collection of classes, functions, and bitmap buttons that you can link into your program for an alternative graphical look to standard Windows controls.

VBX controls

The VBX Library option allows you to use Visual Basic controls in your applications. This option is available for both 16- and 32-bit Windows applications.

CTL3D controls

The CTL3D option lets you use the Windows 3-dimensional controls contained in CTL3DV32.DLL.

Libraries

NewTarget

TargetExpert displays different library options, depending upon the Target Type and Platform you select. Choose which libraries to add to your project or which standard definitions to use if you are building a static library.

- OLE - uses the OLE operating system extensions that let applications achieve a high degree of integration. OLE provides a set of standard interfaces so that one OLE program can interact fully with another OLE program.
- No exceptions - uses the no exceptions library to create executable modules that you can link to non-C++ applications.
- BGI (DOS only) - uses the BGI graphics library file for BGI graphics.

OLE library

NewTarget

OLE (object linking and embedding) is an operating system extension that lets applications achieve a high degree of integration. OLE provides a set of standard interfaces so that any OLE program can interact fully with any other OLE program. No program needs to have any built-in knowledge of its possible partners.

To learn more about the Borland ObjectComponents library and Borland C++ support for OLE, see the ObjectComponents library Help (OCF.HLP).

No exceptions library

NewTarget

The No Exceptions library is a full 16-bit RTL that was created with C++ exception handling turned off. Select this option if you are building a 16-bit target that you plan to use in a non-C++ environment.

For example, use the No Exception library if you are creating .DLL module that will be linked to a Paradox application. This is necessary because the Paradox application does not know how to handle a C++ exception throw().

Borland graphics interface (BGI) library

NewTarget

The BGI Library option controls the automatic searching of the Borland Graphics Interface library. When this option is checked, you can build and run single-file graphics programs without using a project file. Unchecking this option speeds up the link step because the linker doesn't have to search in the BGI graphics library file.

The BGI library is not Windows-compatible. You can uncheck this option and still build programs that use BGI graphics, provided that you add the name of the BGI library (GRAPHICS.LIB) to your project list.

Library versions

NewTarget

You can compile your program using the following framework libraries:

- **Dynamic** - links the dynamic (.DLL) form of the selected libraries. This option allows your application to bind to the standard libraries at run time. The functions in the standard libraries are located in .DLL files rather than directly attached to your application. This greatly reduces the size of an application, but the target is then dependent on the presence of the .DLL libraries at run time. This option forces the large model for 16-bit targets.

  If you create an executable module that uses the dynamic libraries, you will need to supply one or more of the Borland C++ redistributable files with your .EXE or .DLL.

- **Static** - links the static (.LIB) form of the selected libraries. This option causes the standard library functions to be bound directly to your executable file, creating a larger, standalone executable. If you're building many .DLLs with statically bound RTLs, each .DLL gets its own copy of the routines it uses.
- **Diagnostic** - links the diagnostic ObjectWindows or Class Libraries. Diagnostic libraries add debug information to the files. If you select Diagnostic, you must build these libraries. Both OWL and Class Library sources are provided.
- **Multithreaded** - links the 32-bit, multithreaded, flat model run-time library. Multithread is available only if your platform is Win32. By default, applications generated from the IDE are multithread.
- **Alternate Startup** - (DOS applications only) links the alternate startup library module COF*x*.OBJ, which makes SS=DS for all memory models.

Building diagnostic libraries

A diagnostic version of these libraries is supplied on the CD version of Borland C++. If you are using the disk version of the product, you must build a diagnostic version. Source is provided.

To build a diagnostic version of the libraries, use the following make line:

```
MAKE -DDIAGS
```

- See the makefile for other options.

Redistributable files

NewTarget

If you create an executable module that uses the dynamic version of the Borland C++ libraries, you will need to supply one or more of the Borland C++ redistributable .DLL files with your .EXE or .DLL.

To insure that your are including the correct redistrubutable files, use InstallShield Express (supplied with the Development Suite edition of Borland C++) to create a Setup program that goes with your application. InstallShield Express determines the proper .DLL files to include with your application according to the type of application you are building. If you do not have InstallShield Express, refer to REDIST.TXT (located in the DOC directory off your main BCW directory) for more information on the Borland C++ redistributable files.

Math Support (DOS only)

NewTarget

The Math Support options are available only when you are setting options for a DOS target. The following floating-point options are available:

Floating Point - (Command-line equivalent = **-f87**)

Emulation - (Command-line equivalent = **-f**)

No Math Support - (Command-line equivalent = **-f-**)

Floating Point option

`(Command-line equivalent = `**`-f287`**`)`

If your program is going to run only on machines that have an 80287 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80X87 autodetection and emulation logic. This option causes Borland C++ to link your programs with FP287.LIB instead of with EMU.LIB.

Emulation option

`(Command-line equivalent = -f)`

You can use this option for all programs, regardless if they use floating-point math. With the Emulation enabled (the default), the compiler generates code as if the 80X87 were present, but it also links the emulation library (EMU.LIB). When the program runs, it uses the 80X87 if it is present; if no coprocessor is present at run time, the program uses special software that emulates the 80X87. This software uses 512 bytes of your stack, so make allowances for it when using the emulation option and set your stack size accordingly.

Default = ON

No Math Support option

`(Command-line equivalent = -f-)`

If there is no floating-point code in your program, you can save a small amount of link time by choosing No Math Support for the floating-point code-generation option. Then Borland C++ will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

Default = OFF

Transfer Macros

See also

The Borland C++ IDE recognizes certain macro names (or parameters that you want passed to the program) in the Command Line input box (parameter string) of the Tools dialog box. There are three kinds of transfer macros:

- **IDE-state macros** expand to a value that's based on the state of the IDE.
- **File macros** manipulate file names and file-path specifications.
- **Instruction macros** tell the IDE to perform an action or make a setting.

IDE-state macros

$ARG	expands to the command-line parameters to be passed to a program
$COL	expands to the column number of current editor
$CONFIG	expands to the name of the BCW configuration file
$DEF	expands to the contents of the Compiler|Defines for the active node
$DEPLIST()	generates a list of dependencies for the currently selected node
$ENV	expands to the contents of the indicated environment variable
$ERRCOL	expands to the column number of the currently highlighted error in the Message window
$ERRLINE	expands to the line number of the currently highlighted error in the Message window
$ERRNAME	expands to the file name associated with the currently highlighted error in the Message window
$INC	expands to the include path of the current node
$LIB	expands to the library path of the current node
$LINE	expands to the line number in the current Edit window
$PRJNAME	expands to the current loaded project file
$SELNODES	expands to the nodes currently selected in the Project window
$TARGET	expands to the name of the current target

File macros

$DIR()	expands to the directory of file argument
$DRIVE()	expands to the drive of file argument
$EDNAME	expands to the name of file in active editor or project node
$EXT()	expands to the extension of file argument
$NAME()	expands to the root name of the file argument
$OUTNAME	expands to output location of the current node

Instruction macros

$CAP EDIT	redirects the output of a transfer to the Edit window
$CAP MSG()	redirects program output into Message window after being filtered by the indicated script function
$IMPLIB	executes IMPLIB
$NOSWAP	suppresses the display of the running program
$PROMPT	expands to a prompt with parameter string inserted after prompt
$RSP()	places the expanded contents of the macro into a temporary file and name of temporary file is returned

$SAVE ALL	saves the IDE environment, desktop, project, and messages
$SAVE CUR	saves modified file in current Edit window
$SAVE EDIT	saves the modified contents of all open Edit windows
$SAVE PROMPT	same as $SAVE EDIT, except the IDE prompts before saving each open file
$TASM	expands to the command line for running TASM.EXE
$TD	launches the appropriate version of Turbo Debugger for the current target
$TLINKRSP	generates a response file for calling TLINK
$UPTODATE	forces a make on the current target
$WRITEMSG()	copies contents of Message window to specified ASCII file

Alphabetical List of Transfer Macros

See also

$ARG	expands to the command-line parameters to be passed to a program
$CAP EDIT	redirects the output of a transfer to the Edit window
$CAP MSG()	redirects program output into Message window after being filtered by the indicated script function
$COL	expands to the column number of current editor
$CONFIG	expands to the name of the BCW configuration file
$DEF	expands to the contents of the Compiler|Defines for the active node
$DEPLIST()	generates a list of dependencies for the currently selected node
$DIR()	expands to the directory of file argument
$DRIVE()	expands to the drive of file argument
$EDNAME	expands to the name of file in active editor or project node
$ENV	expands to the contents of the indicated environment variable
$ERRCOL	expands to the column number of the currently highlighted error in the Message window
$ERRLINE	expands to the line number of the currently highlighted error in the Message window
$ERRNAME	expands to the file name associated with the currently highlighted error in the Message window
$EXT()	expands to the extension of file argument
$IMPLIB	executes IMPLIB
$INC	expands to the include path of the current node
$LIB	expands to the library path of the current node
$LINE	expands to the line number in the current Edit window
$NAME()	expands to the root name of the file argument
$NOSWAP	suppresses the display of the running program
$OUTNAME	expands to output location of the current node
$PRJNAME	expands to the current loaded project file
$PROMPT	expands to a prompt with parameter string inserted after prompt
$RSP()	places the expanded contents of the macro into a temporary file and name of temporary file is returned
$SAVE ALL	saves the IDE environment, desktop, project, and messages
$SAVE CUR	saves modified file in current Edit window
$SAVE EDIT	saves the modified contents of all open Edit windows
$SAVE PROMPT	same as $SAVE EDIT, except the IDE prompts before saving each open file
$SELNODES	expands to the nodes currently selected in the Project window
$TARGET	expands to the name of the current target
$TASM	expands to the command line for running TASM.EXE
$TD	launches the appropriate version of Turbo Debugger for the current target
$TLINKRSP	generates a response file for calling TLINK
$UPTODATE	forces a make on the current target

$WRITEMSG()   copies contents of Message window to specified ASCII file

Transfer arguments

When you choose a tool that contains a $PROMPT transfer macro in its command line, you will be prompted to enter arguments for the $PROMPT macro.

The position of the $PROMPT macro in the command line determines what you see when prompted for arguments. Anything preceding the $PROMPT macro is treated as a constant and won't be visible.

You can change or add to the parameter string before executing the tool.

ARG

Transfer Macros

The $ARG transfer macro expands to the command-line arguments to be passed to a program run from the IDE using the Debug|Run menu command. You specify command-line arguments from the Debugger page of the Environment Options dialog box (choose Options|Environment).

For example,

```
$TD $ARG
```

runs the stand-alone debugger on the current target, passing the specified command-line arguments to the program.

CAP EDIT

See also Transfer Macros

The $CAP EDIT transfer macro tells the IDE to redirect program output into the Edit window.

After the transfer program is completed, a new Edit window (titled Transfer Output) is created, and the captured output is displayed in that window.

For $CAP EDIT to work correctly:

- the transfer program must write to DOS standard output
- you must also use the $NOSWAP macro.

CAP MSG (filter)

See also Transfer Macros

The $CAP MSG transfer macro captures the program output and places it in the Message window, using a script filter for the converting program output into Message window format.

The following script filters are provided:

- BORL2MSG: The Borland standard output filter can be used with any tool that outputs messages using the Borland error mesage format. Among other tools, this filter works with the Borland C++ compilers (BCC and BCC32), the Borland C++ preprocessors (CPP and CPP32), and Turbo Linker (TLINK and TLINK32).
- GREP2MSG: Grep filter
- RC2MSG: Borland resource compiler (BRC) and MS resource compiler (RC) filter
- TASM2MSG: Turbo Assembler (TASM) and Microsoft assembler (MASM)
- HC312MSG: Microsoft help compiler (HC31) filter
- generic: Generic filter for use with any program that has line-oriented message output

The script source code to these filters is also provided in the SCRIPT subdirectory of your Borland C++ installation directory. If needed, you can write your own filters for other transfer programs using the script source (Filters.spp) as a starting point.

COL

Transfer Macros

The $COL transfer macro expands to the column number of the cursor in the current Edit window.

If the active window is not an Edit window, the string is set to 0.

CONFIG

Transfer Macros

The $CONFIG transfer macro expands to the complete file name of the current configuration file.

This is a null string if no configuration file is defined. This macro is intended for use by programs that access or modify the configuration file. Besides providing the name of the file, this macro causes the current configuration to be saved (if modified) and reloaded when control returns to the IDE.

DEF

Transfer Macros

The $DEF transfer macro expands to the contents of the Compiler|Defines input box.

Use this macro to specify #define directives to an external translator.

DEPLIST( )

See also Transfer Macros

The $DEPLIST([node-type[,prefix[,post-fix]]]) transfer macro expands to a list of dependencies for currently selected node in the Project window.

The list is derived from the supplied extension by searching the dependencies of the selected node for a list of those extensions. Multiple dependencies will be separated by a space. Default project nodes are included in the list, as are nodes in associated source pools. For example,

```
$DEPLIST(.OBJ)
```

when applied to a .DLL or .EXE node, would list all the object nodes (.OBJ files) that would be linked into the target.

In addition, each item in the list can be prepended by an optional prefix and/or postpended by an optional postfix. For example, on a .LIB node, you can type `$DEPLIST (.OBJ, -t)` where `-t` will be prefixed to each .OBJ.

DIR(filename)

Transfer Macros

The $DIR transfer macro expands to the directory portion of the path passed as an argument, without a trailing backslash.

Example

```
$DIR(D:\bc\bin\bcc.exe)
```

expands to:

```
\bc\bin
```

DRIVE(filename)

Transfer Macros

The $DRIVE transfer macro expands to the drive of the file argument.

Example

```
$DRIVE(D:\bc\bin\bcc.exe)
```

expands to:

```
D:
```

EDNAME

Transfer Macros

The $EDNAME transfer macro expands to the complete file name of file in the active Edit window or highlighted node in the Project window.

If no Edit windows or Project windows are active, $EDNAME performs the same function as $SELNODES, expanding to the items currently selected in the Project window.

ENV(variable)

Transfer Macros

The $ENV transfer macro expands to the contents of the environment variable supplied to the macro. If the specified variable cannot be found in the environment, the expansion does nothing.

Example

The macro

```
$ENV(PATH)
```

expands to the path setting in the environment.

ERRCOL

See also Transfer Macros

The $ERRCOL transfer macro expands to the column number of the current error displayed in the Message window.

If there are no messages, the string is expanded to null string.

ERRLINE

See also Transfer Macros

The $ERRLINE transfer macro expands to the line number of the current error selected in the Message window.

If there are no messages, the string is expanded to null string.

ERRNAME

See also Transfer Macros

The $ERRNAME transfer macro expands to the complete file name of file referred to by the selected message in the Message window.

$ERRNAME returns a null string if:

- there are no messages, or
- the current message does not refer to a file.

EXT(filename)

Transfer Macros

The $EXT transfer macro expands to the extension of the file argument; it does not include the period.

Example

The macro

```
$EXT(D:\bc\bin\bcc.exe)
```

expands to

```
exe
```

IMPLIB

Transfer Macros

The $IMPLIB transfer macro executes IMPLIB.

$IMPLIB expands to:

```
$NOSWAP $CAP MSG(IMPL2MSG)$OUTNAME $EDNAME
```

The IMPL2MSG filter captures messages to a file that can be displayed in the Message window.

INC

See also Transfer Macros

The $INC transfer macro expands to the list of include directories defined for the current node.

If a project is not loaded, the global include directory settings are returned.

LIB

See also Transfer Macros

The $LIB transfer macro expands to the list of library directories defined for the current node.

If a project is not loaded, the global library directory settings are returned.

LINE

Transfer Macros

The $LINE transfer macro expands to the line number where the cursor is located in the active Edit window.

If the active window is not an Edit window, the string is set to 0.

NAME(filename)

Transfer Macros

The $NAME transfer macro expands to the file name part of the file argument. It does not include the period.

Example

The macro

```
$NAME(D:\bc\bin\bcc.exe)
```

returns

```
bcc
```

NOSWAP

Transfer Macros

The $NOSWAP transfer macro instructs the IDE to not show the transfer tool while it is running. Windows applications and .PIF files are invoked with SW_HIDE if you run with $NOSWAP. If you run a WINOLDAPP from the IDE, you will notice significant screen swapping if you do not include this macro in the transfer tool command line.

$NOSWAP has no effect on WINOLDAPPS run under STANDARD mode Windows.

Use this macro in conjunction with $CAP_MSG.

OUTNAME

Transfer Macros

The $OUTNAME macro expands to the path and file name that appears in the Intermediate or Final output box (depending on translator). This macro is useful when you are specifying modules for your user-defined translators.

For example, if the project contains STARS.C, the default Output Path file name is STARS.OBJ.

If there is no project defined, $OUTNAME performs the same function as $EDNAME.

PRJNAME

See also Transfer Macros

The $PRJNAME macro expands to the name of the current project file.

The default project BCWDEF.BCW is returned if no project is loaded.

PROMPT

See also Transfer Macros

The $PROMPT macro tells the IDE to display the transfer program's expanded parameter string in the Program Arguments input box for verification. You can change or add to the string before it is passed to the transfer program.

The position of $PROMPT in the command line determines what is shown in the dialog box. You can place constant parameters in the command line by placing them before $PROMPT.

Example

In the following macro statement,

```
/C $PROMPT dir
```

the `/C` is constant and does not show in the dialog box, but `dir` is displayed and can be edited before the command is run.

RSP(contents)

Transfer Macros

The $RSP macro places the expanded contents of the macro into a temporary file and the name of the temporary file is returned. The IDE attempts to place the temporary file as follows:

1. in the project directory
2. Windows temporary directory
3. the current directory if it differs from the project directory

The temporary file will automatically be deleted when the transfer to the tool is completed.

SAVE ALL

See also Transfer Macros

The $SAVE ALL transfer macro tells the IDE to the current state of the environment, desktop, project and Message window. This macro is equivalent to choosing the Options|Save command with all the Save options checked.

SAVE CUR

See also Transfer Macros

The $SAVE CUR transfer macro tells the IDE to save the file in the current Edit window if it has been modified. This ensures that the transfer program will use the latest version of the source file.

SAVE EDIT

See also Transfer Macros

The $SAVE EDIT macro saves all files that are open in the IDE Edit windows.

SAVE PROMPT

See also Transfer Macros

The $SAVE PROMPT transfer macro saves all files open in Edit windows, prompting you before it saves each one. This macro is similar to the $SAVE EDIT macro.

SELNODES

See also Transfer Macros

The $SELNODES transfer macro expands to the nodes currently selected in the Project window. Each node name will be separated by spaces.

TARGET

See also Transfer Macros

The $TARGET transfer macro expands to the target name of the currently selected node in the Project window.

TASM

Transfer Macros

The $TASM transfer macro is predefined for use with Turbo Assembler (TASM). It uses the TASM2MSG filter to trap TASM messages. $TASM is essentially shorthand for the following:

```
$NOSWAP $SAVE CUR $CAP MSG(TASM2MSG.DLL) /ml $EDNAME , $OUTNAME
```

TD

Transfer Macros

The $TD transfer macro launches the appropriate version Turbo Debugger for the current target. The appropriate debugger is invoked with the result of $TARGET.

TLINKRSP

See also   Transfer Macros

The $TLINKRSP transfer macro builds a response file for calling TLINK. The following macro statement designates the use of TLINK as the executable:

```
@$RSP($TLINKRSP)
```

UPTODATE

Transfer Macros

The $UPTODATE transfer macro forces a make on the current target before invoking the transfer tool.

WRITEMSG(filename)

Transfer Macros

The $WRITEMSG() transfer macro copies the contents of the Message window to the specified ASCII file.

The translator can parse the file and act on the desired messages. For example:

```
WRITEMSG(C:\MESSAGES.TXT)
```

writes to the file MESSAGES.TXT in your root directory.

16-bit compiler options (Options|Project)

The 16-bit Compiler options affect the compilation of all 16-bit source modules. It is usually best to keep the default setting for most options in this section.

The subtopics are

Processor

Calling convention

Memory model

Segment names data

Segment names far data

Segment names code

Entry/Exit code

16-bit compiler | Processor (Options|Project|16-bit Compiler)

The Processor options let you specify the minimum CPU type compatible with your program. These options introduce instructions specific to the CPU type you select to increase performance.

The options are

Instruction set

Data alignment

16-bit instruction set (Options|Project|16-bit Compiler|Processor)

The Instruction Set options specify for which CPU instruction set the compiler should generate code.

8086

`(Command-line equivalent: -1-)`

Choose the 8086 option if you want the compiler to generate 16-bit code for the 8086-compatible instruction set. (To generate 8086 code, you must not turn on the options **-2**, **-3**, **-4**, or **-5**.)

80186

`(Command-line equivalent: -1)`

Choose the 80186 option if you want the compiler to generate extended 16-bit code for the 80186 instruction set. Also supports the 80286 running in Real mode.

80286

`(Command-line equivalent: -2)`

Choose the 80286 option if you want the compiler to generate 16-bit code for the 80286 protected-mode-compatible instruction set.

80386

`(Command-line equivalent: -3)`

Choose the 80386 option if you want the compiler to generate 16-bit code for the 80386 protected-mode-compatible instruction set.

i486

`(Command-line equivalent: -4)`

Choose the i486 option if you want the compiler to generate 80386/i486 instructions running in enhanced-mode Windows.

Default = BCC.EXE, 80286 (-2); IDE, 80386 (-3)

Data alignment (Options|Project|16-bit Compiler|Processor or 32-bit Compiler|Processor)

The Data Alignment options let you choose the compiler aligns data in stored memory. Word, double-word, and quad-word alignment forces integer-size and larger items to be aligned on memory addresses that are a multiple of the type chosen. Extra bytes are inserted in structures to ensure that members align correctly.

Byte alignment

`(Command-line equivalent: -a1 or -a-)`

When Byte Alignment is turned on, the compiler does not force alignment of variables or data fields to any specific memory boundaries; the compiler aligns data at either even or odd addresses, depending on which is the next available address.

While byte-wise alignment produces more compact programs, the programs tend to run a bit slower. The other data alignment options increase the speed that 80x86 processors fetch and store data.

Word alignment (2-byte)

`(Command-line equivalent: -a2)`

When Word Alignment is on, the compiler aligns non-character data at even addresses. Automatic and global variables are aligned properly. **char** and **unsigned char** variables and fields can be placed at any address; all others are placed at an even-numbered address.

Double word (4-byte)

`(Command-line equivalent: -a4, 32-bit only)`

Double Word alignment aligns non-character data at 32-bit word (4-byte) boundaries.

Quad word (8-byte)

`(Command-line equivalent: -a8, 32-bit only)`

Quad Word alignment aligns non-character data at 64-bit word (8-byte) boundaries.

Default = Byte Alignment (**-a-**)

16-bit Calling Conventions (Options|Project|16-bit Compiler)

Calling Convention options tell the compiler which calling sequences to generate for function calls. The C, Pascal, and Register calling conventions differ in the way each handles stack cleanup, order of parameters, case, and prefix of global identifiers.

You can use the **_ _cdecl**, **_ _pascal**, or **_ _fastcall** keywords to override the default calling convention on specific functions.

- These options should be used by experts only.

C

(Command-line equivalent: **-pc, -p-**)

This option tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the **_ _cdecl** keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed).

Pascal

(Command-line equivalent: **-p**)

This option tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the **_ _pascal** keyword. The resulting function calls are usually smaller and faster than those made with the C (**-pc**) calling convention. Functions must pass the correct number and type of arguments.

Register

(Command-line equivalent: **-pr**)

This option forces the compiler to generate all subroutines and all functions using the **Register** parameter-passing convention, which is equivalent to declaring all subroutine and functions with the **_ _fastcall** keyword. With this option enabled, functions or routines expect parameters to be passed in registers.

Default = C (**-pc**)

16-bit compiler | Memory model (Options|Project|16-bit Compiler)

The Memory Model section lets you specify the organization of segments for code and data in your 16-bit programs. (32-bit programs always use the flat memory model.) Large is the most common memory model used for Windows programs. All .OBJ and .LIB files in your program should be compiled in the same memory model.

The options are

Model

Assume SS equals DS

Options

Put constant strings in code segments

Far virtual tables

Automatic far data

Fast huge pointers

Far data threshold

Model (Options|Project|16-bit Compiler|Memory Model)

The Model options specify the memory model you want to use. The memory model you choose determines the default method of memory addressing.

Tiny

`(Command-line equivalent: -mt ,DOS only)`

This is the smallest of the memory models. Use this model when memory is at an absolute premium. All four segment registers (CS, DS, SS, ES) are set to the same address. You have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the **/t** option.

Small

`(Command-line equivalent: -ms)`

Use the small model for average size applications. The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

The **-ms!** command-line option compiles using the small model and assumes DS != SS. To achieve this in the IDE, you need to check both the **Small** and **Never** options.

Medium

`(Command-line equivalent: -mm)`

Use the medium model for large programs that do not keep much data in memory. Far pointers are used for code but not for data. Data and stack together are limited to 64K, but code can occupy up to 1 MB.

The **-mm!** command-line option compiles using the medium model and assumes DS != SS. To achieve this in the IDE, you need to check both the **Meduim** and **Never** options.

- The net effect of the **-ms!** and **-mm!** options is actually very small. If you take the address of a stack variable (parameter or auto), the default (DS == SS) is to make the resulting pointer a near (DS relative) pointer. This way, you can assign the address to a default-sized pointer in those models without problems. When DS != SS, the pointer type created when you take the address of a stack variable is an **_ss** pointer. This means that the pointer can be freely assigned or passed to a far pointer or to an **_ss** pointer. But for the memory models affected, assigning the address to a near or default-sized pointer produces a "Suspicious pointer conversion" warning. Such warnings are usually errors.

Compact

`(Command-line equivalent: -mc)`

Use the compact model if your code is small but you need to address a lot of data. The Compact model is the opposite of the medium model: far pointers are used for data but not for code; code is limited to 64K, pointers can point almost anywhere. All functions are near by default and all data pointers are far by default.

Large

`(Command-line equivalent: -ml)`

Use the large model for very large applications only. Far pointers are used for both code and data. Data is limited to 1MB. Far pointers can point almost anywhere. All functions and data pointers are far by default.

Huge

`(Command-line equivalent: -mh, DOS only)`

Use the huge model for very large applications only. Far pointers are used for both code and data. Borland C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

Default = Large in IDE; Small in BCC.EXE

Assume SS equals DS (Options|Project|16-bit Compiler|Memory Model)

The Assume SS Equals DS options specify how the compiler considers the stack segment (SS) and the data segment (DS).

Default for memory model

The memory model you use determines whether the stack segment (SS) is equal to the data segment (DS) Usually, the compiler assumes that SS is equal to DS in the small and medium memory models (except for DLLs).

Never

`(Command-line equivalent: -Fs-)`

The compiler assumes that the SS is never equal to DS. This is always the case in the compact and large memory models and when building a Windows DLL.

Always (DOS only)

`(Command-line equivalent: -Fs)`

The compiler always assumes that SS is equal to DS in all memory models. This option causes the compiler to use the alternate C0F*x*.OBJ startup module (which places the stack in the data segment) instead of C0*x*.OBJ. You can use this option when porting code originally written for an implementation that makes the stack part of the data segment.

Default = Default for Memory Model

Put constant strings in code segments (Options|Project|16-bit Compiler|Memory Model)

(Command-line equivalent: **-dc**)

This option moves all string literals from the data segment to the code segment of the generated object file, making the data type **const**.

- Use this option only with compact or large memory models. In addition, this option does not work with overlays.

Using this option saves data segment space. In large programs, especially those with a large number of literal strings, this option shifts the burden from the data segment to the code segment.

Default = OFF

Far virtual tables (Options|Project|16-bit Compiler|Memory Model)

(Command-line equivalent: **-Vf**)

When you turn this option on, the compiler creates virtual tables in the code segment instead of the data segment, unless you override this option using the Far Virtual Tables Segment (**-zV**) or Far Virtual Tables Class (**-zW**) options. Virtual table pointers are made into full 32-bit pointers (which is done automatically if you are using the huge memory model).

You can use Far Virtual Tables to remove the virtual tables from the data segment (which might be getting full). You might also use this option to share objects (of classes with virtual functions) between modules that use different data segments (for example, a DLL and an executable that uses that DLL).

You must compile all modules that might share objects entirely with or entirely without this option.

- You can get the same effect by using the **huge** or **_export** modifiers on a class-by-class basis.

This option changes the mangled names of C++ objects.

Default = OFF

Fast huge pointers (Options|Project|16-bit Compiler|Memory Model)

(Command-line equivalent: **-h**)

This option offers an alternative method of calculating huge pointer expressions. This option behaves differently for DOS and Windows programs.

For Windows programs, huge pointers are normalized by the value of the variable _ALTINCR, which is initialized by Windows at the startup time of the application.

For DOS programs, this option offers a faster method of "normalizing" than the standard method. (*Normalizing* is resolving a memory address so that the offset is always less than 16.) When you use this option, huge pointers are normalized only when a segment wraparound occurs in the offset part, which causes problems with huge arrays if an array element crosses a segment boundary.

Usually, Borland C++ normalizes a huge pointer whenever adding or subtracting from it. This ensures, for example, that if you have an array of **struct**s that's larger than 64K, indexing into the array and selecting a **struct** field always works with **struct**s of any size. Borland C++ accomplishes this by always normalizing the results of huge pointer operations--the address offset contains a number that is no higher than 15 and a segment wraparound never occurs with huge pointers. The disadvantage of this approach is that it tends to be quite expensive in terms of execution speed.

Default = OFF

Automatic far data (Options|Project|16-bit Compiler|Memory Model)

(Command-line equivalent: **-Ff**)

When the Automatic Far Data option is enabled, the compiler automatically places data objects larger than or equal to the threshold size into far data segments. The threshold size defaults to 32,767. This option is useful for code that doesn't use the huge memory model, but declares enough large global variables that their total size is close to or exceeds 64K. This option has no effect for programs that use tiny, small, and medium memory models.

When this option is disabled, the size value is ignored

This option and the Far Data Threshold input box work together. The Far Data Threshold specifies the minimum size above that which data objects will be automatically made far.

If you use this option with the Generate COMDEFs option (**-Fc**), the COMDEFs become far in the compact, large, and huge models.

Default = OFF

The command-line option **-Fm** enables all the other **-F** options (**-Fc**, **-Ff**, and **-Fs**). You can use **-Fm** as a handy shortcut when porting code from other compilers. To do this in the IDE, check the Automatic Far Data and Always options on this Project Options page, and the Generate COMDEFs option on the Compiler|Floating Point page.

Far data threshold (Options|Project|16-bit Compiler|Memory Model)

(Command-line equivalent: **-Ff=*size***, where ***size*** = threshold size)

Use Far Data Threshold to specify the size portion needed to complete the Automatic Far Data option.

Default = 32767 (if Automatic Far Data is disabled, this option value is ignored)

16-bit compiler | Segment names data (Options|Project|16-bit Compiler)

Use Segment Names Data to change the default segment, group, and class names for initialized and uninitialized data.

- Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

The options available are

Initialized Data

Uninitialized Data

Initialized Data (Options|Project|16-bit Compiler|Segment Names Data)

Use Initialized Data to change the default segment, group, and class names for initialized data.

In all options, use an asterisk (*) for ***name*** to select the default segment names.

▪ Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Initialized data segment

(Command-line equivalent = **-zR*name***)

Sets the name of the initialized data segment to *name*. By default, the initialized data segment is named _DATA for **near** data and *modulename*_DATA for **far** data.

Initialized data group

(Command-line equivalent = **-zS*name***)

Sets the name of the initialized data segment group to *name*. By default, the data group is named DGROUP.

Initialized data class

(Command-line equivalent = **-zT*name***)

Sets the name of the initialized data segment to *name*. By default, the initialized data segment class is named DATA.

Default = * (default segment name) for all options

Uninitialized data (Options|Project|16-bit Compiler|Segment Names Data)

Use Uninitialized Data to change the default segment, group, and class names for code uninitialized data.

In all options, use an asterisk (*) for ***name*** to select the default segment names.

- Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Uninitialized data (BSS segment)

(Command-line equivalent = **-zD*name***)

Sets the name of the uninitialized data segment. By default, the uninitialized data segment is named _BSS for **near** uninitialized data and *modulename*_BBS for **far** uninitialized data.

Uninitialized data (BSS group)

(Command-line equivalent = **-zG*name***)

Sets the name of the uninitialized data segment group t *name*. By default, the data group is named DGROUP.

Uninitialized data (BSS class)

(Command-line equivalent = **-zB*name***)

Sets the name of the uninitialized data segment class to *name*. By default, the uninitialized data segments are assigned to class BSS.

Default = * (default segment name) for all options

16-bit compiler | Segment names far data (Options|Project|16-bit Compiler)

16-bit Compiler|Segment Names Far Data options set the far data segment name, group, class name, and the far virtual tables segment name and class.

The options are

Far data
Far data segment
Far data group
Far data class

Far virtual tables
Virtual table segment
Virtual table class

Far data (Options|Project|16-bit Compiler|Segment Names Far Data)

Use Far Data to change the default segment, group, and class names for far data.

In all options, use an asterisk (*) for ***name*** to select the default segment names.

- Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Far data segment

(Command-line equivalent = **-zE*name***)

Sets the name of the segment where **_ _far** objects are placed to *name*. By default, the segment name is the name of the far object followed by _DATA.

Far data group

(Command-line equivalent = **-zH*name***)

Causes **_ _far** objects to be placed into the group *name*. By default, far objects are not placed into a group.

Far data class

(Command-line equivalent = **-zF*name***)

Sets the name of the class for **_ _far** objects to *name*. By default, the name is FAR_DATA.

Default = * (default segment name) for all options

Far virtual tables (Options|Project|16-bit Compiler|Segment Names Far Data)

Use Far Virtual Tables to change the default segment and class names virtual tables.

In all options, use an asterisk (*) for ***name*** to select the default segment names.

- Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Virtual table segment

(Command-line equivalent = **-zV*name***)

Sets the name of the **_ _far** virtual table segment to *name*. By default, far virtual tables are generated in the CODE segment.

Virtual table class

(Command-line equivalent = **-zW*name***)

Sets the name of the far virtual table class segment to *name.* By default, far virtual table classes are generated in the CODE segment.

Default = * (default segment name) for all options

16-bit compiler | Segment names code (Options|Project|16-bit Compiler)

Segment Names Code options let you specify a new code segment name and reassign the group and class.

The options are

Code segment

Code group

Code class

Code (Options|Project|16-bit Compiler|Segment Names Code)

Use Code to change the name of the code segment as well as the code group and class.

In all options, use an asterisk (*) for ***name*** to select the default segment names.

- Do not change the settings in this dialog box unless you are an expert.

Code segment

(Command-line equivalent = **-zC*name***)

Sets the name of the code segment to *name*. By default, the code segment is named _CODE for near code and *modulename*_TEXT for far code, except for the medium and large models where the name is *filename*_CODE (*filename* is the source file name).

Code group

(Command-line equivalent = **-zP*name***)

Causes any output files to be generated with a code group for the code segment named *name*.

Code class

(Command-line equivalent = **-zA*name***)

Changes the name of the code segment class to *name*. By default, the code segment is assigned to class CODE.

Default = * (default segment name) for all options

16-bit compiler | Entry/Exit code (Options|Project|16-bit Compiler)

Entry/Exit Code options tell the compiler what type of application the compiler creates by specifying what prolog and epilog code to generate.

- Although these options are listed in the 16-bit compiler section, these options also apply to 32-bit programs.

The options are

Windows all functions exportable

Windows explicit functions exported

Windows smart callbacks

Windows smart callbacks, explicit functions exportable

Windows DLL, all functions exportable

Windows DLL, Explicit Functions Exported

Entry/Exit code (Options|Project|16-bit Compiler|Entry/Exit Code)

These options specify which type of prolog and epilog code the compiler generates for each module's functions.

Windows all functions exportable

(Command-line equivalent: **-tW**)

This option creates a Windows object function prolog/epilog for all **_ _far** functions, then sets up those functions to be called from another module. This option assumes that all functions can be called by the Windows kernel or by other modules and generates the necessary overhead information for every **_ _far** function (whether the function needs it or not).

To export the function address from the .EXE to a .DLL, the code includes a call to *MakeProcInstance()*, passing the resulting pointer to the .DLL that requested the address of the function. For the function to be exportable, the function must be declared as **_export** or the function name must be included in the .DEF file of the executable.

This option creates the most general Windows executable, but not necessarily the most efficient.

- The **-W** command-line option is supported for backward compatibility, and is equivalent to **-tW**.

Default = ON

Windows explicit functions exported

(Command-line equivalent: **-tWE**)

This option creates a Windows object module in which only **_ _far** functions declared as **_export** functions are exportable. Use this option if you have functions that will not be called by the Windows kernel. Windows Explicit Functions Exported operates the same as Windows All Functions Exportable except that only those functions marked with the **_export** keyword (and methods of classes marked as **_export**) are given the extra prolog/epilog.

This option is far more efficient than Windows All Functions Exportable, since only those functions called from outside the module get the prolog overhead. This option requires that you determine in advance which functions or classes need to be exported. *MakeProcInstance()* is still used, but no .DEF file manipulation is needed (if you use this options along with **_export**, you don’t need to define exports in your .DEF files).

- The **-WE** command-line option is supported for backward compatibility, and is equivalent to **-tWE**.

Default = OFF

Windows smart callbacks, all functions exportable

(Command-line equivalent: **-tWS**)

This option creates an object module with smart callbacks for all **_ _far** functions exported. Use this option only if the compiler can assume that DS == SS for all functions in the module (which is true for the vast majority of Windows programs and the default for Borland tools).

This option creates a Windows .EXE function prolog/epilog for all **_ _far** functions and sets them up to be called from another module. *MakeProcInstance()* does not need to be called and you do not need to edit the .DEF file.

- The **-WS** command-line option is supported for backward compatibility, and is equivalent to **-tWS**.

Default = OFF

Windows smart callbacks, explicit functions exportable

(Command-line equivalent: **-tWSE**)

This option is the same as Windows Smart Callbacks except that only those functions marked with the **_export** keyword (and methods of classes marked as **_export**) are given the extra prolog/epilog. This is efficient since only those functions called from outside the module get the prolog overhead. This option requires determining in advance which functions/classes need to be exported.

- The **-WSE** command-line option is supported for backward compatibility, and is equivalent to **-tWSE**.

Default = OFF

Windows DLL, all functions exportable

(Command-line equivalent: **-tWD**)

This option creates a Windows .DLL function prolog/epilog for all **_ _far** functions, then sets up those functions to be called from another module. To actually export function addresses from the .DLL, the functions must be marked with the **_export** keyword or the function names need to be included in the .DEF file of the executable.

- The **-WD** command-line option is supported for backward compatibility, and is equivalent to **-tWD**.

Default = OFF

Windows DLL, explicit functions exported

(Command-line equivalent: **-tWDE**)

This option is the same as Windows DLL, All Functions Exportable except that only **_ _far** functions marked with the **_export** keyword (and methods of classes marked as **_export**) are given the extra prolog/epilog. This is far more efficient than the “Windows DLL, All Functions Exportable” option since only those functions called from outside the module get the prolog overhead. This option requires determining which functions or classes need to be exported in advance. No .DEF file manipulation is needed.

- The **-WDE** command-line option is supported for backward compatibility, and is equivalent to **-tWDE**.

Default = OFF

32-bit compiler options (Options|Project|32-bit Compiler)

The 32-bit Compiler page contains two radio buttons that allow you to select which 32-bit compiler backend you want to use when compiling 32-bit applications.

Borland optimizing compiler

If you are compiling from the command line, use BCC32.EXE.

Note: With Borland's new 32-bit compiler, you cannot pass a temporary variable by reference. You must pass it by const reference.

32-bit compiler options

32-bit compiler options listed on the Processor and Calling Convention pages affect the compilation of all 32-bit Windows applications for Windows NT and Windows 95 . Because 32-bit programs use a flat memory model (they are not segmented), there are fewer options to configure than for 16-bit programs.

The subtopics are

Processor

Calling conventions

32-bit Compiler | Processor (Options|Project|32-bit Compiler|Processor)

The 32-bit Compiler Processor options specify which CPU instruction set to use and how to handle floating-point code for 32-bit programs.

The options are

Instruction set

Data alignment

32-bit instruction set (Options|Project|32-bit Compiler|Processor)

The Instruction Set options specify for which CPU instruction set the compiler should generate code.

80386

`(Command-line equivalent: -3)`

Choose the 80386 option if you want the compiler to generate 80386 protected-mode compatible instructions running on Windows 95 or Windows NT. This is the default setting.

i486

`(Command-line equivalent: -4)`

Choose the i486 option if you want the compiler to generate i486 protected-mode compatible instructions running on Windows 95 or Windows NT.

Pentium

`(Command-line equivalent: -5)`

Choose the Pentium option if you want the compiler to generate Pentium instructions on Windows 95 or Windows NT.

While this option increases the speed at which the application runs on Pentium machines, expect the program to be a bit larger than when compiled with the **80386** or **i486** options. In addition, **Pentium**-compiled code will sustain a performance hit on non-Pentium systems.

Default = 80386 (-3)

32-bit calling conventions (Options|Project|32-bit Compiler|Calling Convention)

Calling Convention options tell the compiler which calling sequences to generate for function calls. The C, Pascal, and Register calling conventions differ in the way each handles stack cleanup, order of parameters, case, and prefix of global identifiers.

You can use the **_ _cdecl**, **_ _pascal**, **_ _fastcall,** or **_ _stdcall** keywords to override the default calling convention on specific functions.

- These options should be used by experts only.

C

`(Command-line equivalent: -pc, -p-)`

This option tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the **_ _cdecl** keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed).

You can use the **_ _pascal**, **_ _fastcall**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Pascal

`(Command-line equivalent: -p)`

This option tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the **_ _pascal** keyword. The resulting function calls are usually smaller and faster than those made with the C (`-pc`) calling convention. Functions must pass the correct number and type of arguments.

You can use the **_ _cdecl**, **_ _fastcall**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Register

`(Command-line equivalent: -pr)`

This option forces the compiler to generate all subroutines and all functions using the **Register** parameter-passing convention, which is equivalent to declaring all subroutine and functions with the **_ _fastcall** keyword. With this option enabled, functions or routines expect parameters to be passed in registers.

You can use the **_ _pascal**, **_ _cdecl**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Standard Call (32-bit Compiler Only)

`(Command-line equivalent: -ps)`

This option tells the compiler to generate a Stdcall calling sequence for function calls (does not generate underscores, preserve case, called function pops the stack, and pushes parameters right to left). This is the same as declaring all subroutines and functions with the **_ _stdcall** keyword. Functions must pass the correct number and type of arguments.

You can use the **_ _cdecl**, **_ _pascal**, **_ _fastcall** keywords to specifically declare a function or subroutine using another calling convention.

Default = C (`-pc`)

Build Attributes (Options|Project|Build Attributes)

See Also

Build attributes affect whether or not a node is built during compilation. The icons associated with each of these options are displayed next to the nodes in the Project hierarchy diagram. Build attributes are set in the Options|Project dialog box.

- This set of options is not available for an AppExpert project.

The options are

Always build

Check Always Build and the node is always built, even if it has not changed.

Build when out of date

Check Build When Out of Date and the node is built only if it has changed.

Never build

Check Never Build and the node is not built.

Can't build

Check Can't Build to be notified when a node cannot be built.

Exclude from parent

Check Exclude from Parent and the system indicates when a node should be excluded from parent (such as with source pools).

Compiler options (Options|Project)

Compiler options are common to all C and C++ programs. They directly affect how the compiler generates code.

The subtopics are

Defines

Code generation

Floating point

Compiler output

Source

Debugging

Precompiled headers

Compiler | Defines (Options|Project|Compiler)

(Command-line equivalent: **-D*name*** and **-D*name*=*string***)

The macro definition capability of Borland C++ lets you define and undefine macros (also called *manifest* or *symbolic* constants) in the IDE or on the command line. The macros you define override those defined in your source files.

- You can use the $INHERIT and $ENV( ) macros to specify the defines for the project node you are modifying.

Defining macros from the IDE

Preprocessor definitions (such as those used in **#if** statements and macro definitions) can be entered on the Compiler Defines page. The following rules apply when using the Defines input box:

- Separate multiple definitions with semicolons (;), and assign values with an equal sign (=). For example:

```
Switch1;Switch2;Switch3=OFF
```

- Leading and trailing spaces are stripped, but embedded spaces are left intact.
- If you want to include a semicolon in a macro, precede the semicolon with a backslash (\).

Defining macros on the command line

On the command line, the **-D*name*** option defines the identifier *name* to the null string. **-D*name*=*string*** defines *name* to *string*. In this assignment, *string* cannot contain spaces or tabs. You can also define multiple **#define** options on the command line using either of the following methods:

- Include multiple definitions after a single **-D** option by separating each define with a semicolon (;) and assigning values with an equal sign (=). For example:

```
BCC.EXE -Dxxx;yyy=1;zzz=NO MYFILE.C
```

- Include multiple **-D** options, separating each with a space. For example:

```
BCC.EXE -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C
```

Compiler | Code generation (Options|Project|Compiler)

Compiler Code Generation options affect how code is generated.

The options are

Allocate enums as ints
Unsigned characters
Duplicate strings merged
fastthis
Register variables

Allocate enums as ints (Options|Project|Compiler|Code Generation)

(Command-line equivalent: **-b**)

When the Allocate Enums As Ints option is on, the compiler always allocates a whole word (a two-byte **int** for 16-bits or a four-byte **int** for 32-bits) for enumeration types (variables of type **enum**).

When this option is off (**-b-**), the compiler allocates the smallest integer that can hold the enumeration values: the compiler allocates an **unsigned** or **signed char** if the values of the enumeration are within the range of 0 to 255 (minimum) or -128 to 127 (maximum), or an **unsigned** or **signed short** if the values of the enumeration are within the following ranges:

- 0 to 65,535 (minimum) or -32,768 to 32,767 (maximum) (16-bit)
- 0 to 4,294,967,295 or -2,147,483,648 to 2,147,483,647 (32-bit)

The compiler allocates a two-byte **int** (16-bit) or a four-byte **int** (32-bit) to represent the enumeration values if any value is out of range.

Default = ON

Unsigned characters (Options|Project|Compiler|Code Generation)

(Command-line equivalent: **-K**)

When the Unsigned Characters option is on, the compiler treats all **char** declarations as if they were **unsigned char** type, which provides compatibility with other compilers.

Default = OFF (**char** declarations default to **signed**; **-K-**)

Duplicate strings merged (Options|Project|Compiler|Code Generation)

(Command-line equivalent: **-d**)

When you check the Duplicate Strings Merged option, the compiler merges two literal strings when one matches another. This produces smaller programs (at the expense of a slightly longer compile time), but can introduce errors if you modify one string.

Default = OFF (**-d-**)

fastthis (Options|Project|Compiler|Code Generation)

(Command-line equivalent: **-po**, 16-bit only)

This option causes the compiler to use the **_ _fastthis** calling convention when passing the **this** pointer to member functions. The **this** pointer is passed in a register (or a register pair in 16-bit large data models). Likewise, calls to member functions load the register (or register pair) with **this**. Note that you can use **_ _fastthis** to compile specific functions in this manner.

When **this** is a 'near' (16-bit) pointer, it is supplied in the SI register; for 'far' **this** pointers, DS:SI is used. If necessary, the compiler saves and restores DS. All references in the member function to member data are done via the SI register.

The names of member functions compiled with **_ _fastthis** are mangled differently from non-fastthis member functions, to prevent mixing the two. It is easiest to compile all classes with **_ _fastthis**, but you can compile some classes with **_ _fastthis** and some without, as in the following example:

```
// no -po on the command-line
class X;
#pragma option -po
class Y      //Y will use fastthis
{
...
};
class X      //X will not use fastthis,
{            //since its class declaration
             //appeared before fastthis was turned on
...
};
#pragma option -po-
```

- If you use a makefile to build a version of the class library that has **_ _fastthis** enabled, you must define `_CLASSLIB_ALLOW_po` and use the **-po** option. The `_CLASSLIB_ALLOW_po` macro can be defined in <Your_BCW_dir>\INCLUDE\SERVICES\borlandc.h
- If you use a makefile to build a **_ _fastthis** version of the runtime library, you must define `_RTL_ALLOW_po` and use the **-po** option.
- If you rebuild the libraries and use **-po** without defining the appropriate macro, the linker emits undefined symbol errors.

Default = OFF

Register variables (Options|Project|Compiler|Code Generation)

These options suppress or enable the use of register variables.

None

`(Command-line equivalent: -r-)`

Choose None to tell the compiler not to use register variables even if you have used the **register** keyword.

Register keyword

`(Command-line equivalent: -rd)`

Choose Register Keyword to tell the compiler to use register variables only if you use the **register** keyword and a register is available. Use this option or the Automatic option (**-r**) to optimize the use of registers.

- You can use **-rd** in **#pragma** options.

Automatic

`(Command-line equivalent: -r)`

Choose Automatic to tell the compiler to automatically assign register variables if possible, even when you do not specify a register variable by using the **register** type specifier.

Generally, you can keep this option set to Automatic unless you are interfacing with preexisting assembly code that does not support register variables.

Default = Automatic (**-r**)

Compiler | Floating point (Options|Project|Compiler)

Floating Point options tell the compiler how to handle floating-point code and floating-point optimization.

The options are

Floating point
No floating point
Fast floating point

Floating point (Options|Project|Compiler|Floating Point)

The Floating Point options specify how the compiler handles floating-point numbers in your code.

No floating point

(Command-line equivalent: **-f-**)

Choose No Floating Point if you are not using floating point. No floating-point libraries are linked when this option is enabled (**-f-**). If you enable this option and use floating-point calculations in your program, you will get link errors. When unchecked (**-f**), the compiler emulates 80x87 calls at runtime.

Default = OFF   (**-f**)

Fast floating point

(Command-line equivalent: **-ff**)

When Fast Floating Point is on, floating-point operations are optimized without regard to explicit or implicit type conversions. Calculations can be faster than under ANSI operating mode.

When this option is unchecked (**-ff-**), the compiler follows strict ANSI rules regarding floating-point conversions.

Default = OFF

Correct Pentium FDIV flaw

(Command-line equivalent: **-fp**)

Some early Pentium chips do not perform specific floating-point division calculations with full precision. Although your chances of encountering this problem are slim, this switch inserts code that emulates floating-point division so that you are assured of the correct result. This option decreases your program's FDIV instruction performance.

- Use of this option only corrects FDIV instructions in modules that you compile. The run-time library also contains FDIV instructions which are not modified by the use of this switch. To correct the run-time libraries, you must recompile them using this switch.

The following functions use FDIV instructions in assembly language which are not corrected if you use this option:

acos	**cosh**	**pow10l**
acosl	**coshl**	**powl**
asin	**cosl**	**sin**
asinl	**exp**	**sinh**
atan	**expl**	**sinhl**
atan2	**fmod**	**sinl**
atan2l	**fmodl**	**tan**
atanl	**pow**	**tanh**
cos	**pow10**	**tanhl**
tanl		

In addition, this switch does not correct functions that convert a floating-point number to or from a string (such as *printf* or *scanf*).

Default = OFF

Compiler | Compiler output (Options|Project|Compiler)

Set control of object file contents on the Compiler Output page.

The options are

Autodependency information
Generate underscores
Generate COMDEFs

Autodependency information (Options|Project|Compiler|Compiler Output)

(Command-line equivalent: **-X-**)

When the Autodependency option is checked (**-X-**), the compiler generates autodependency information for all project files with a .C or .CPP extension.

The Project Manager can use autodependency information to speed up compilation times. The Project Manager opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file when the source module is compiled. After that, the time and date of every file that was used to build the .OBJ file is checked against the time and date information in the .OBJ file. The source file is recompiled if the dates are different. This is called an autodependency check.

If the project file contains valid dependency information, the Project Manager does the autodependency check using that information. This is much faster than reading each .OBJ file.

When this option is unchecked (**-X**), the compiler does not generate the autodependency information.

Modules compiled with autodependency information can use MAKE's autodependency feature.

Default = ON (**-X-**)

Generate underscores (Options|Project|Compiler|Compiler Output)

(Command-line equivalent: **-u**)

When the Generate Underscores option is on, the compiler automatically adds an underscore character (_) in front of every global identifier (functions and global variables) before saving them in the object module. Pascal identifiers (those modified by the **_ _pascal** keyword) are converted to uppercase and are not prefixed with an underscore.

Underscores for C and C++ are optional, but you should turn this option on to avoid errors if you are linking with the standard Borland C++ libraries.

Default = ON

Generate COMDEFs (Options|Project|Compiler|Compiler Output)

`(Command-line equivalent: **-Fc**, 16-bit only)`

Generate COMDEFs generates communal variables (COMDEFs) for global C variables that are not initialized and not declared as **static** or **extern**. Use this option when header files included in several source files contain global variables.

For example, a definition such as

```
int SomeArray[256];
```

could appear in a header file that is then included in many modules. When this option is on, the compiler generates *SomeArray* as a communal variable rather than a public definition (a COMDEF record rather than a PUBDEF record). You can use this option when porting code that uses a similar feature with another implementation.

The linker generates only one instance of the variable, so it will not be a duplicate definition linker error. As long as a given variable does not need to be initialized to a nonzero value, you do not need to include a definition for it in any of the source files.

Default = OFF

Compiler | Source (Options|Project|Compiler)

Compiler|Source options set source code interpretation.

The options are:

Source

Nested comments

Identifier Length

Language compliance

Borland extensions

ANSI

UNIX V

Kernighan and Ritchie

Compatibility

MFC compatibility

Nested comments (Options|Project|Compiler|Source)

(Command-line equivalent: **-C**)

When the Nested Comments option is on, you can nest comments in your C and C++ source files.

Nested comments are not allowed in standard C implementations, and they are not portable.

Default = OFF

Identifier length (Options|Project|Compiler|Source)

(Command-line equivalent: **-i*n***, where ***n*** = significant characters)

Use the Identifier Length input box to specify the number of significant characters (those which will be recognized by the compiler) in an identifier.

Except in C++, which recognizes identifiers of unlimited length, all identifiers are treated as distinct only if their significant characters are distinct. This includes variables, preprocessor macro names, and structure member names.

Valid numbers for *n* are 0, and 8 to 250, where 0 means use the maximum identifier length of 250.

By default, Borland C++ uses 250 characters per identifier. Other systems (including some UNIX compilers) ignore characters beyond the first eight. If you are porting to other environments, you might want to compile your code with a smaller number of significant characters, which helps you locate name conflicts in long identifiers that have been truncated.

Default = 250

Language compliance (Options|Project|Compiler|Source)

The Language Compliance options tell the compiler how to recognize keywords in your programs.

Borland extensions

`(Command-line equivalents: -A-, -AT)`

The Borland Extensions option tells the compiler to recognize Borland's extensions to the C language keywords, including **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **_export**, **_ds**, **_cs**, **_ss**, **_es**, and the register pseudovariables (**_AX**, **_BX**, and so on). For a complete list of keywords, see the keyword index.

ANSI

`(Command-line equivalent: -A)`

The ANSI option compiles C and C++ ANSI-compatible code, allowing for maximum portability. Non-ANSI keywords are ignored as keywords.

UNIX V

`(Command-line equivalent: -AU)`

The UNIX V option tells the compiler to recognize only UNIX V keywords and treat any of Borland's C++ extension keywords as normal identifiers.

Kernighan and Ritchie

`(Command-line equivalent: -AK)`

The Kernighan and Ritchie option tells the compiler to recognize only the K&R extension keywords and treat any of Borland's C++ extension keywords as normal identifiers.

Hint: If you get declaration syntax errors from your source code, check that this option is set to Borland Extensions.

Default = Borland Extensions (`-A-`)

MFC compatibility (Options|Project|Compiler|Source)

See also

(Command-line equivalents: **-VF**)

Turn this option on to compile code that is compatible with the Microsoft foundation classes (MFC). Among other things, the compiler makes the following adjustments to be compatible with MFC:

- Accepts spurious semicolons in a class scope
- Allows anonymous structs
- Uses the old-style scoping resolution in **for** loops
- Allows methods to be declared with a calling convention, but leaves off the calling convention in the definition
- Tries the operator **new** if it cannot resolve a call to the operator **new[ ]**
- Lets you omit the operator & on member functions
- Allows a const class that is passed by value to be treated as a trivial conversion, not as a user conversion
- Allows you to use a cast to a member pointer as a selector for overload resolution, even if the qualifying type of the member pointer is not derived from the class in which the member function is declared
- Accepts declarations with duplicate storage in a class, as in
  `extern "C" typedef`
- Accepts and ignores `#pragma comment(linker, "...")` directives

Default = OFF

Compiler | Debugging (Options|Project|Compiler)

See also

Compiler Debugging options affect the generation of debug information during compilation. When linking larger .OBJ files, you may need to turn these options off to increase the available system resources.

The options are

Standard stack frame

Test stack overflow

Out-of-line inline functions

Line numbers

Generate debug information

Detach debug information from OBJs

Browser reference information in OBJs

Standard stack frame (Options|Project|Compiler|Debugging)

See also

(Command-line equivalent: **-k**)

When the Standard Stack Frame option is on, the compiler generates a standard stack frame (standard function entry and exit code). This is helpful when debugging, since it simplifies the process of stepping through the stack of called subroutines.

When this option is off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code smaller and faster.

The Standard Stack Frame option should always be on when you compile a source file for debugging.

Default = ON

Test Stack Overflow (Options|Project|Compiler|Debugging)

See also

`(Command-line equivalent: -N, 16-bit only)`

When the this option is on, the compiler generates stack overflow logic at the entry of each function.

Even though this is costly in terms of both program size and speed, it can be a real help when trying to track down difficult stack overflow bugs. If an overflow is detected, the run-time error message `Stack overflow!` is generated, and the program exits with an exit code of 1.

- Stack overflow testing is always enabled in the 32-bit compilers (this adds a minimal overhead to 32-bit programs).

Default = OFF

Out-of-line inline functions (Options|Project|Compiler|Debugging)

See also

(Command-line equivalent: **-vi**)

When the Out-of-Line Inline Functions option is on, the compiler expands C++ inline functions inline.

To control the expansion of inline functions, the Generate debug Information option (**-v**) acts slightly different for C++ code: when inline function expansion is disabled, inline functions are generated and called like any other function.

Because debugging with inline expansion can be difficult, the command-line compilers provide the following options:

- **-v** turns debugging on and inline expansion off
- **-v-** turns debugging off and inline expansion on
- **-vi** turns inline function expansion on
- **-vi-** turns inline expansion off (inline functions are expanded out of line)

For example, if you want to turn both debugging and inline expansion on, use the **-v** and **-vi** options.

Default = OFF

Line numbers (Options|Project|Compiler|Debugging)

See also

(Command-line equivalent: **-y**)

When the Line Numbers option is on, the compiler automatically includes line numbers in the object and object map files. Line numbers are used by both the integrated debugger and Turbo Debugger.

Although the Generate debug information option (**-v**) automatically generates line number information, you can turn that option off (**-v-**) and turn on Line Numbers (**-y**) to reduce the size of the debug information generated. With this setup, you can still step, but you will not be able to watch or inspect data items.

Including line numbers increases the size of the object and map files but does not affect the speed of the executable program.

- When Line Numbers is on, make sure you turn off Jump Optimization in the 16-bit specific optimizations and Pentium scheduling in the 32-bit Compiler options. When these options are enabled, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = OFF

Generate debug information (Options|Project|Compiler|Debugging)

See also

(Command-line equivalent: **-v**)

When this option is checked, debug information will be included in your .OBJ files. The compiler passes this option to the linker so it can include the debugging information in the .EXE file. For debugging, this option treats C++ inline functions as normal functions.

You need debugging information to use either the integrated debugger or the standalone Turbo Debugger.

When this option is off (**-v-**), you can link and create larger object files. While this option does not affect execution speed, it does affect compilation and link time.

- When Line Numbers is on, make sure you turn off Jump Optimization in the 16-bit specific optimizations and Pentium scheduling in the 32-bit Compiler options. When these options are enabled, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = ON

Detach debug information from OBJs (Options|Project|Compiler|Debugging)

See also

`(Command-line equivalent: -He)`

When Detach debug Information from OBJs is on, the compiler puts debug information into a separate file instead of collecting it in the OBJ file.

This option speeds up link time. The resulting file has the same name as the CPP file name, but has the extension .#xx, where xx is a number. For example,

```
FILENAME.#00
```

Default = OFF

Browser reference information in OBJs (Options|Project|Compiler|Debugging)

See also

(Command-line equivalent: **-R**)

When the Browser Info In OBJs option is on, the compiler generates additional browser-specific information such as location and reference information. This information is then included in your .OBJ files. In addition to this option, you need debugging information (**-v**) to use the Browser.

When this option is off, you can link and create larger object files. While this option does not affect execution speed, it does affect compilation time and program size.

Default = ON

Compiler | Precompiled headers (Options|Project|Compiler)

Using precompiled header files can dramatically increase compilation speed by storing an image of the symbol table on disk in a file, then later reloading that file from disk instead of parsing all the header files again. Directly loading the symbol table from disk is much faster than parsing the text of header files, especially if several source files include the same header file.

The options are

Generate and use

Use but do not generate

Do not generate or use

Cache precompiled header

Precompiled header name

Stop precompiling after header file

- You can use the $INHERIT and $ENV( ) macros in any of the precompiled header input fields.

Precompiled headers (Options|Project|Compiler|Precompiled Headers)

See also

Using precompiled headers can dramatically increase compilation speeds, though they require a considerable amount of disk space.

Generate and use

(Command-line equivalent: **-H**)

When this option is enabled, the IDE generates and uses precompiled headers. The default file name is <*projectname*>.CSM for IDE projects and BCDEF.CSM (16-bit) or BC32DEF.CSM (32-bit) for the command-line compilers.

Use but do not generate

(Command-line equivalent: **-Hu**)

When the Use But Do Not Generate option is on, the compilers use preexisting precompiled header files; new precompiled header files are not generated.

Do not generate or use

(Command-line equivalent: **-H-**)

When the Do Not Generate Or Use option is on, the compilers do not generate or use precompiled headers.

Default = Do not generate or use (**-H-**)

Cache precompiled header (Options|Project|Compiler|Precompiled Headers)

(Command-line equivalent: **-Hc**)

When you enable this option, the compiler caches the precompiled headers it generates. This is useful when you are precompiling more than one header file.

- To use this option, you must also enable the Generate and Use (**-H**) precompiled header option.

Default = OFF

Precompiled header name (Options|Project|Compiler|Precompiled Headers)

(Command-line equivalent: **-H=*filename***)

This option lets you specify the name of your precompiled header file. The compilers set the name of the precompiled header to *filename*.

When this option is enabled, the compilers generate and use the precompiled header file that you specify.

Stop precompiling after header file (Options|Project|Compiler|Precompiled Header)

(Command-line equivalent: **-H"*xxx*"**; for example -H"owl/owlpch.h")

This option terminates compiling the precompiled header after the compiler compiles the file specified as *xxx*. You can use this option to reduce the amount of disk space used by precompiled headers.

When you use this option, the file you specify must be included from a source file for the compiler to generate a .CSM file.

- You cannot specify a header file that is included from another header file. For example, you cannot list a header included by windows.h because this would cause the precompiled header file to be closed before the compilation of windows.h was competed.

C++ options (Options|Project|C++ Options)

Project|C++ Options affect compilation of all C and C++ programs. For most of the C++ options, you'll usually want to use the default settings.

The subtopics are

Member pointer

C++ compatibility

Virtual tables

Templates

Exception handling/RTTI

General

C++ options | Member pointer (Options|Project|C++ Options|Member Pointers)

Use C++ Member Pointers options to direct member pointers and affect how the compiler treats explicit casts.

The options are

Honor precision of member pointers

Member pointer representation

Support all cases

Support multiple inheritance

Support single inheritance

Smallest for class

Honor precision of member pointers (Options|Project|C++ Options|Member Pointers)

(Command-line equivalent: **-Vmp**)

When this option is enabled, the compiler uses the declared precision for member pointer types. Use this option when a pointer to a derived class is explicitly cast as a pointer-to-member of a simpler base class (when the pointer is actually pointing to a derived class member).

Default = OFF

Member pointer representation (Options|Project|C++ Options|Member Pointers)

The C++ Member Pointers options specify what member pointers can point to.

Support all cases

`(Command-line equivalent: -Vmv)`

When this option is enabled, the compiler places no restrictions on where member pointers can point. Member pointers use the most general (but not always the most efficient) representation.

Default = ON

Support multiple inheritance

`(Command-line equivalent: -Vmm)`

When this option is enabled, member pointers can point to members of multiple inheritance classes (with the exception of virtual base classes).

Default = OFF

Support single inheritance

`(Command-line equivalent: -Vms)`

When this option is enabled, member pointers can point only to members of base classes that use single inheritance.

Default = OFF

Smallest for class

`(Command-line equivalent: -Vmd)`

When this option is enabled, member pointers use the smallest possible representation that allows member pointers to point to all members of their particular class. If the class is not fully defined at the point where the member pointer type is declared, the most general representation is chosen by the compiler and a warning is issued.

Default = OFF

C++ options | C++ compatibility (Options|Project|C++ Options|C++ Compatibility)

Use the C++ Compatibility options to handle C++ compatibility issues, such as handling 'char' types, specifying options about hidden pointers, passing class arguments, adding hidden members and code to a derived class, passing the 'this' pointer to 'Pascal' member functions, changing the layout of classes, or insuring compatibility when class instances are shared with non-C++ code or code compiled with previous versions of Borland C++.

Don't restrict scope of 'for' loop expression variables

Do not treat 'char' as distinct type

Calling convention mangling compatibility

Destructor cleanup compatibility

Class layout compatibility

Virtual base pointers

Always near

Same size as 'this' pointer

Options

Pass class values via reference to temporary

Disable constructor displacements

Push 'this' first for pascal member functions

'deep' virtual bases

Vtable pointer follows data members

Treat 'far' classes as 'huge'

Don't restrict scope of 'for' loop expression variables (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent: **-Vd**)

This option lets you specify the scope of variables declared in **for** loop expressions. The output of the following code segment changes, depending on the setting of this option.

```
int main(void)
{

  for(int i=0; i<10; i++)
  {
        cout << "Inside for loop, i = " << i << endl;
  }     //end of for-loop block

  cout << "Outside for loop, i = " << i << endl;  //error without -Vd

}       //end of block containing for loop
```

If this option is disabled (the default), the variable *i* goes out of scope when processing reaches the end of the **for** loop. Because of this, you'll get an Undefined Symbol compilation error if you compile this code with this option disabled.

If this option is enabled (**-Vd**), the variable *i* goes out of scope when processing reaches the end of the block containing the **for** loop. In this case, the code output would be:

```
Inside for loop, i = 0
...
Outside for loop, i = 10
```

Default = OFF

Do Not Treat 'char' as Distinct Type (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent: **-K2**, 16-bit)

Allow only signed and unsigned char types. The Borland C++ compiler allows for signed char, unsigned char, and char data types. This option treats **char** as signed.

This option is provided for compatibility with previous versions of Borland C++ (3.1 and earlier) and supports only 16-bit programs.

Default = OFF

Calling convention mangling compatibility (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent: **-VC**)

When this option is enabled, the compiler disables the distinction of function names where the only possible difference is incompatible code generation options. For example, with this option enabled, the linker will not detect if a call is made to a **_ _fastcall** member function with the **cdecl** calling convention.

This option is provided for backward compatibility only; it lets you link old library files that you cannot recompile.

Default = OFF

Destructor cleanup compatibility (Options|Project|C++ Options|C++ Compatibility)

When this option is enabled, non-Borland DLLs can operate in conjunction with a Borland EXE that has exception handling turned on. This is a switch for backward compatibility.

Default = OFF

Class layout compatibility (Options|Project|C++ Options|C++ Compatibility)

When this option is enabled, OBJ files from previous versions of Borland C++ can be linked with the new linker.

Default = OFF

Virtual base pointers (Options|Project|C++ Options|C++ Compatibility)

When a class inherits virtually from a base class, the compiler stores a hidden pointer in the class object to access the virtual base class subobject.

The Virtual Base Pointers options specify options about the hidden pointer.

Always near

(Command-line equivalent: **-Vb-**)

When the Always Near option is on, the hidden pointer will always be a near pointer. (When a class inherits virtually from a base class, the compiler stores a *hidden pointer* in the class object to access the virtual base class subobject.)

This option allows for the smallest and most efficient code.

Same size as 'this' pointer

(Command-line equivalent: **-Vb**)

When the Same Size as ‘this' Pointer option is on, the compiler matches the size of the hidden pointer to the size of the **this** pointer in the instance class.

This allows for compatibility with previous versions of the compiler.

Default = Always Near (**-Vb-**)

Pass class values via reference to temporary (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent: **-Va**)

When this option is enabled, the compiler passes class arguments using the "reference to temporary" approach. When an argument of type class with constructors is passed by value to a function, this option instructs the compiler to create a temporary variable at the calling site, initialize this temporary variable with the argument value, and pass a reference from this temporary to the function.

This option insures compatibility with previous versions of the compiler.

Default = OFF

Disable constructor displacements (Options|Project|C++ Options|C++ Compatibility)

`(Command-line equivalent: -Vc)`

When the Disable Constructor Displacements option is enabled, the compiler does not add hidden members and code to a derived class (the default).

This option insures compatibility with previous versions of the compiler.

Default = OFF

Push 'this' first for Pascal member functions **(Options|Project|C++ Options|C++ Compatibility)**

`(Command-line equivalent: -Vp)`

When this option is enabled, the compiler passes the **this** pointer to Pascal member functions as the first parameter on the stack.

By default, the compiler passes the **this** parameter as the last parameter on the stack, which permits smaller and faster member function calls.

Default = OFF

'deep' virtual bases (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent: **-Vv**)

When a derived class overrides a virtual function which it inherits from a virtual base class, and a constructor or destructor for the derived class calls that virtual function using a pointer to the virtual base class, the compiler can sometimes add hidden members to the derived class. These “hidden members” add code to the constructors and destructors.

This option directs the compiler *not* to add the hidden members and code so that the class instance layout is the same as with previous version of Borland C++; the compiler does not change the layout of any classes to relax the restrictions on pointers.

Default = OFF

Vtable pointer follows data members (Options|Project|C++ Options|C++ Compatibility)

`(Command-line equivalent` **`-Vt`**`)`

When this option is enabled, the compiler places the virtual table pointer after any nonstatic data members of the specified class.

This option insures compatibility when class instances are shared with non-C++ code and when sharing classes with code compiled with previous versions of Borland C++.

Default = OFF

Treat 'far' classes as 'huge' (Options|Project|C++ Options|C++ Compatibility)

(Command-line equivalent **-Vh**)

When this option is enabled, the compiler treats all classes declared **_ _far** as if they were declared as **_ _huge**. For example, the following code normally fails to compile. Checking this option allows the following code fragment to compile:

```
struct __huge A
{
virtual void f();  // A vtable is required to see the error.
};
struct __far B  :  public A
{
};
// Error: Attempting to derive a far class from the huge base 'A'.
```

Default = OFF

C++ Options | Virtual Tables (Options|Project|C++ Options)

C++ Options|Virtual Tables options control C++ virtual tables and the expansion of inline functions when debugging.

The options are

Linkage

Smart

Local

External

Public

Virtual tables linkage (Options|Project|C++ Options|Virtual Tables)

The C++ Virtual Tables options control C++ virtual tables and the expansion of inline functions when debugging.

Smart

(Command-line equivalent: **-V**)

This option generates common C++ virtual tables and out-of-line inline functions across the modules in your application. As a result, only one instance of a given virtual table or out-of-line inline function is included in the program.

The Smart option generates the smallest and most efficient executables, but produces .OBJ and .ASM files compatible only with TLINK and TASM.

Default = ON

Local

(Command-line equivalent: **-Vs**)

You use the Local option to generate local virtual tables (and out-of-line inline functions) so that each module gets its own private copy of each virtual table or inline function it uses.

The Local option uses only standard .OBJ and .ASM constructs, but produces larger executables.

Default = OFF

External

(Command-line equivalent: **-V0**)

You use the External option to generate external references to virtual tables. If you don't want to use the Smart or Local options, use the External and Public options to produce and reference global virtual tables.

- When you use this option, one or more of the modules comprising the program must be compiled with the Public option to supply the definitions for the virtual tables.

Default = OFF

Public

(Command-line equivalent: **-V1**)

Public produces public definitions for virtual tables. When using the External option (**-V0**), at least one of the modules in the program must be compiled with the Public option to supply the definitions for the virtual tables. All other modules should be compiled with the External option to refer to that Public copy of the virtual tables.

Default = OFF

C++ options | Templates (Options|Project|C++ Options)

Use the options under C++ Options | Templates to tell the compiler how to generate template instances in C++.

The options are

Instance Generation

Smart

Global

External

Templates instance generation (Options|Project|C++ Options|Templates)

See also

The Template Instance Generation options specify how the compiler generates template instances in C++.

Smart

`(Command-line equivalent: -Jg)`

When the Smart option is enabled, the compiler generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker automatically merges duplicates to produce a single copy of the instance.

To generate the instances, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class), typically in a header file.

This is a convenient way of generating template instances.

Default = ON

Global

`(Command-line equivalent: -Jgd)`

When the Global option is on, the compiler generates public (global) definitions for all template instances.

The Global option does not merge duplicates. If the same template instance is generated more than once, the linker reports public symbol re-definition errors.

Default = OFF

External

`(Command-line equivalent: -Jgx)`

When the External option is on, the compiler generates external references to all template instances.

When you use this option, all template instances in your code must be publicly defined in another module with the external option (**-Jgd**) so that external references are properly resolved.

Default = OFF

C++ options | Exception handling/RTTI (Options|Project|C++ Options)

Use the Exceptions Handling options to enable or disable exception handling and to tell the compiler how to handle the generation of run-time type information.

If you use exception handling constructs in your code and compile with exceptions disabled, you'll get an error.

The options are

Enable exceptions

Enable exception location information

Enable destructor cleanup

Enable fast exception prologs

Enable compatible exceptions

Enable run-time type information

Enable exceptions (Options|Project|C++ Options|Exception Handling/RTTI)

(Command-line equivalent: **-x**)

When this option is enabled, C++ exception handling is enabled. If this option is disabled (**-x-**) and you attempt to use exception handling routines in your code, the compiler generates error messages during compilation.

Disabling this option makes it easier for you to remove exception handling information from programs; this might be useful if you are porting your code to other platforms or compilers.

- Disabling this option turns off only the compilation of exception handling code; your application can still include exception code if you link .OBJ and library files that were built with exceptions enabled (such as the Borland standard libraries).

Default = ON

Exception handling (Options|Project|C++ Options|Exception Handling/RTTI)

Enable exception location information

(Command-line equivalent: **-xp**)

When this option is enabled, run-time identification of exceptions is available because the compiler provides the file name and source-code line number where the exception occurred. This enables the program to query file and line number from where a C++ exception was thrown.

Default = OFF

Enable destructor cleanup

(Command-line equivalent: **-xd**)

When this option is enabled and an exception is thrown, destructors are called for all automatically declared objects between the scope of the catch and throw statements.

In general, when you enable this option, you should also set Enable Runtime Type Information (**-RT**) as well.

- Destructors are not automatically called for dynamic objects allocated with **new**, and dynamic objects are not automatically freed.

Default = ON

Enable fast exception prologs

(Command-line equivalent: **-xf**)

When this option is enabled, inline code is expanded for every exception handling function. This option improves performance at the cost of larger executable file sizes.

- If you select both Fast Exception Prologs and Enable Compatible Exceptions (**-xc**), fast prologs will be generated but Enable Compatible Exceptions will be disabled (the two options are not compatible).

Default = OFF

Enable compatible exceptions

(Command-line equivalent: **-xc**, 16-bit only)

This option allows .EXEs and .DLLs built with Borland C++ to be compatible with executables built with other products. When Enable Compatible Exceptions is disabled, some exception handling information is included in the .EXE, which could cause compatibility issues.

- Libraries that can be linked into .DLLs need to be built with this option enabled.

Default = OFF

Enable run-time type information **(Options|Project|C++ Options|Exception Handling/RTTI)**

(Command-line equivalent: **-RT**)

This option causes the compiler to generate code that allows run-time type identification.

In general, if you set Enable Destructor Cleanup (**-xd**), you will need to set this option as well.

Default = ON

C++ options | General (Options|Project|C++ Options)

Zero-length empty base classes

`(Command-line equivalent: -Ve)`

Usually the size of a class is at least one byte, even if the class does not define any data members. When this option is enabled, the compiler ignores this unused byte for the memory layout and the total size of any derived classes.

Default = OFF

Zero-length empty class member functions

`(Command-line equivalent: -Vx)`

Usually the size of a data member in a class definition is at least one byte. When this option is enabled, the compiler allows a truly empty structure of zero length.

Default = OFF

Directories options (Options|Project|Directories)

Options|Project

The Directories options tell the Borland C++ compiler where to find or where to put header files, library files, source code, output files, and other program elements.

The subtopics are

Source Directories

Output Directories

Source directories (Options|Project|Directories)

See also

The Source Directories options let you specify the directories that contain your standard include files, library and .OBJ files, and program source files.

Click the down-arrow icon or press Alt+Down arrow to display the history list of previously entered directory names.

- You can use the $INHERIT and $ENV( ) macros in any of the input fields on this page.

Include

(Command-line equivalent: **-I*path***, where *path* = directory path)

Use the Include list box to specify the drive and/or directories that contain program include files. Standard include files are those given in angle brackets (<>) in an #include statement (for example, `#include <myfile>`).

- The Borland compilers and linkers use specific file search algorithms to locate the files needed to complete the compilation and link cycles.

Library

(Command-line equivalent: **-L*path***, where *path* = directory path)

Use the Library list box to specify the directories that contain the Borland C++ IDE startup object files (C0*x*.OBJ), run-time library files (.LIB files), and all other .LIB files. By default, the linker looks for them in the directory containing the project file (or in the current directory if you're using the command-line compiler).

- You can also use the linker option ***/Lpath*** to specify the library search directories when you link files from the command line.

Source

The Source list box specifies the directories where the compiler and the integrated debugger should look for your project source files.

Specifying multiple directories

Multiple directory names are allowed in each of the list boxes; use a semicolon (;) to separate the specified drives and directories. To display a history list of previously entered directory names, click the down-arrow icon or press Alt+Down arrow.

From the command line, you can enter multiple include and library directories in the following ways:

- You can stack multiple entries with a single **-L** or **-I** option by separating directories with a semicolon:

```
BCC.EXE -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
BCC.EXE -Ldirname1 -Ldirname2 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix listings:

```
BCC.EXE -Ldirname1;dirname2 -Iinc1 -Ld:dirname3 -Iinc2;inc3 myfile.c
```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative; the compiler searches all the directories listed in order from left to right.

Guidelines for entering directory names

See Also

Use the following guidelines when entering directories in the Directories options pages.

- You must separate multiple directory path names (if allowed) with a semicolon (;).
- You can use up to a maximum of 127 characters (including whitespace).
- Whitespace before and after the semicolon is allowed but not required.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

For example,

```
C:\;C:..\BC5;D:\myprog\source
```

$INHERIT and $ENV( )

The IDE supports the two macros $INHERIT and $ENV( ) in the Directories page, the Compiler|Defines page and the Compiler|Precompiled Header page of the Project Options dialog box.

- You can add $INHERIT and $ENV( ) anywhere in the strings you type into the input boxes.

$INHERIT

The $INHERIT macro expands to the value of the respective option of the current nodes parent.

For example, suppose the project node MYSOURCE.CPP has a parent node MYSOURCE.EXE, and the defines for MYSOURCE.EXE are

```
WIN31;
```

If you set the Defines value for MYSOURCE.CPP to:

```
_RTLDLL;$INHERIT;STRICT
```

MYSOURCE.CPP will inherit the defines of MYSOURCE.EXE, which will give it the following Defines values:

```
_RTLDLL;WIN31;;STRICT
```

$ENV( )

The $ENV(*environment_variable*) macro expands to the defined value of the specified environment variable. For example, suppose the environment variable BCROOT is set to the following value:

```
BCROOT = C:\BC5
```

You can then set the Include path in the Directories page as follows:

```
$ENV(BCROOT)\Include
```

This will set the actual include path to:

```
C:\BC5\Include
```

File search algorithms

See also

#include-file search algorithms

Borland C++ searches for files included in your source code with the **#include** directive in the following ways:

- If you specify a path and/or directory with your include statement, Borland C++ searches only the location specified. For example, if you have the following statement in your code:

```
#include "c:\bc\include\owl\owl.h"
```

the header file owl.h must reside in the directory C:\BC\INCLUDE\OWL. In addition, if you use the statement:

```
#include <owl\owl.h>
```

and you set the Include option (**-I**) to specify the path `c:\bc\include`, the file owl.h must reside in C:\BC\INCLUDE\OWL, and not in C:\BC\INCLUDE or C:\OWL.

- If you put an `#include <somefile>` statement in your source code, Borland C++ searches for "somefile" only in the directories specified with the Include (**-I**) option.
- If you put an `#include "somefile"` statement in your code, Borland C++ first searches for "somefile" in the current directory; if it does not find the file there, it then searches in the directories specified with the Include (**-I**) option.

Library file search algorithms

The library file search algorithms are similar to those for include files:

- Implicit libraries: Borland C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for `#include <somefile>`.

  Implicit library files are the ones Borland C++ automatically links in and the start-up object file (C0x.OBJ). To see these files in the Project Manager, turn on run-time nodes (choose Options|Environment|Project View, then check Show Runtime Nodes).

- Explicit libraries: Where Borland C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name. Explicit library files are ones you list on the command line or in a project file; these are file names with a .LIB extension.
- If you list an explicit library file name with no drive or directory (like this: `mylib.lib`), Borland C++ first searches for that library in the current directory. If the first search is unsuccessful, Borland C++ looks in the directories specified with the Library (**-L**) option. This is similar to the search algorithm for `#include "somefile"`.
- If you list a user-specified library with drive and/or directory information (like this: `c:\mystuff\mylib1.lib`), Borland C++ searches only in the location you explicitly listed as part of the library path name and not in any specified library directories.

Output directories (Options|Project|Directories)

See Also

The Output Directories options specify the directories where your .OBJ, .EXE, .DLL, and .MAP files are placed. The Borland C++ IDE looks for those directories when performing a make or run and to check dates and times of .OBJs, .EXEs, and .DLLs. If the entry is blank, the files are stored in the current directory.

Click the down-arrow icon or press Alt+Down arrow to display the history list of previously entered directory names.

- You can use the $INHERIT and $ENV( ) macros in any of the input fields on this page.

Intermediate

Use the Intermediate list box to specify where Borland C++ places object (.OBJ) and map (.MAP) files when it builds your project. This is also the directory where a tool (such as Resource Workshop) places any temporary files that it might create.

Final

(Command-line equivalent: **-n*path*,** where *path* = directory path)

The Final list box specifies the location where the IDE places the generated target files (for example, .EXE and .DLL files).

Librarian Options (Options|Project|Librarian)

Librarian options affect the behavior of the built-in librarian. The built-in librarian combines the .OBJ files in your project into .LIB files. Options in this section control that process. In addition, you can cause the librarian to generate a list (.LST) file containing the .OBJs in a generated .LIB and the functions those .OBJs contain.

TLIB.EXE is the command-line librarian.

The options are

Generate list file

Case sensitive library

Purge comment records

Create extended dictionary

Library page size

Generate list file (Options|Project|Librarian)

When the Generate List File option is on, the librarian automatically produces a list file (.LST) that lists the contents of your library when it is created.

Case-sensitive library (Options|Project|Librarian)

(Command-line equivalent = **/C**)

When the Case-Sensitive Library option is on, the librarian treats case as significant in all symbols in the library. For example, if Case-Sensitive Library is checked, "CASE", "Case", and "case" are all treated as different symbols.

Purge comment records (Options|Project|Librarian)

(Command-line equivalent = **/0**)

When the Purge Comment Records option is on, the librarian removes all comment records from modules added to the library.

Create extended dictionary (Options|Project|Librarian)

(Command-line equivalent = **/E**)

When the Create Extended Dictionary option is on, the librarian includes, in compact form, additional information that helps the linker process library files faster.

Library page size (Options|Project|Librarian)

(Command-line equivalent = ***/Psize***, where *size* is number of pages)

The Library Page Size input box is where you set the number of bytes in each library "page" (dictionary entry).

The page size determines the maximum size of the library. Page size must be a power of 2 between 16 and 32,768 inclusive. The default page size of 16 allows a library of about 1 MB in size.

To create a larger library, change the page size to the next higher value (32).

Linker options (Options|Project|Linker)

Linker options affect how an application is linked.

Linker options let you control how intermediate files (.OBJ, .LIB, and .RES) are combined into executables (.EXE) and dynamic-link libraries (.DLL). For most options in this section, you will usually want to keep the default settings.

The subtopics are

General

Map file

16-bit linker

16-bit optimizations

32-bit linker

32-bit image

Warnings

Linker | General (Options|Project|Linker|General)

Use the Linker|General options to include or exclude debugging information from your .EXE or .DLL. Debug information must be included in your program if you want to use the debugger (you can turn it off for production versions).

The options are

Case-sensitive link

Case-sensitive exports and imports (16-bit only)

Include debug information

Ignore default libraries

Pack code segments (16-bit only)

Code pack size

Subsystem version (major.minor)

Case-sensitive link (Options|Project|Linker|General)

(Command-line equivalent = **/c**)

When the Case-Sensitive Link option is enabled, the linker differentiates between upper and lower-case characters in public and external symbols. Normally, this option should be checked, since C and C++ are both case-sensitive languages.

Default = ON

Case-sensitive exports and imports (Options|Project|Linker|General)

(Command-line equivalent = **/C**, 16-bit only)

When the Case-Sensitive Exports option is on, the linker is case sensitive when it processes the names in the IMPORTS and EXPORTS sections of the module definition file.

Use this option when you are trying to export non-callback functions from DLLs, as in exported C++ member functions or dynamic versions of ObjectWindows Library and BIDS.

Do not use this option for normal Windows callback functions (declared FAR PASCAL).

Default = OFF

Include debug information (Options|Project|Linker|General)

(Command-line equivalent = **/v**)

When the Include Debug Information option is on, the linker includes information in the output file needed to debug your application with the Borland C++ Integrated Debugger or Turbo Debugger.

On the command line, this option causes the linker to include debugging information in the executable file for all object modules that contain debugging information. You can use the **/v+** and **/v-** options to selectively enable or disable debugging information on a module-by-module basis (but not on the same command-line where you use **/v**). For example, the following command includes debugging information for modules `mod2` and `mod3`, but not for `mod1` and `mod4`:

```
TLINK mod1 /v+ mod2 mod3 /v- mod4
```

Default = ON in IDE; OFF on the command line

Ignore default libraries (Options|Project|Linker|General)

(Command-line equivalent = **/n**)

When you are linking with modules created by a compiler other than the Borland C++ compiler, the other compiler might have placed a list of default libraries in the object file.

When the Default Libraries option is unchecked (off), the linker tries to find any undefined routines in these libraries and in the default libraries supplied by the C++ IDE.

When this option is checked (on), the linker searches only the default libraries supplied by the C++ IDE and ignores any defaults in .OBJ files. You might want to check this option when linking modules written in another language.

Default = ON

Pack code segments (Options|Project|Linker|General)

(Command-line equivalent = **/P**)

Pack Code Segments has different meanings for 16-bit and 32-bit applications. In addition, Code Segment Packing applies only to Windows applications and DLLs.

For 16-bit links, Code Segment Packing causes the linker to minimize the number of code segments by packing as many code segments as possible into one physical segment up to (and never greater than) the code-segment packing limit, which is set to 8,192 (8K) by default. TLINK starts a new segment if needed.

Because there is a certain amount of system overhead for every segment maintained, code segment packing typically increases performance by reducing the number of segments.

For 32-bit links, Code Packing Segments means the linker packs all code into one “segment.” On the command-line, **/P-** turns this option off.

Default = ON

Code pack size (Options|Project|Linker|General)

(Command-line equivalent = **/P=*n*,** 16-bit only)

Use Code Pack Size to change the default code-packing size to any value between 1 and 65,536. (On the command line, set `n` to a value between 1 and 65,536.)

You would probably want the limit to be a multiple of 4K under the 386 enhanced mode because of the paging granularity of the system. Although the optimum segment size in 386 enhanced mode is 4K, the default code segment packing size is 8K because typical code segments are from 4K to 8K in size, and the default of 8K might pack more efficiently.

Code segment packing typically increases performance because each maintained segment requires system overhead. On the command-line, **/P-** turns code segment packing off, which can be useful if you've turned it on in the configuration file, but want to turn it off for a particular link.

Default = 8192 bytes (8K)

Subsystem version (major.minor) (Options|Project|Linker|General)

(Command-line equivalent = ***/Vd.d***)

This option lets you specify the Windows version ID on which you expect your application will be run. The linker sets the Subsystem version field in the .EXE header to the number you specify in the input box.

You can also set the Windows version ID in the SUBSYSTEM portion of the module definition file (.DEF file) However, any version setting you specify in the IDE or on the command line overrides the setting in the .DEF file.

Command-line usage

When you use the ***/Vd.d*** command-line option, the linker sets the Windows version ID to the number specified by ***d.d***. For example, if you specify /V4.0, the linker sets the Subsystem version field in the .EXE header to 4.0, which indicates a Windows 95 application.

Default = 3.1

Linker | Map file (Options|Project|Linker|Map File)

Linker|Map File options tell the linker what type of map file to produce. You specify the type of map file created with the Map File options. These options control the information generated on segment ordering, segment sizes, and public symbols.

The options are

Off

Segments

Publics

Print mangled names in map file

Include source line numbers

Map file (Options|Project|Linker|Map File)

You use the Map File options to choose the type of map file to be produced at link time.

For settings other than Off, the map file is placed in the output directory defined in the Directories|Output page

Off

(Command-line equivalent = **/x**)

The Off option tells the linker not to create a map file.

Default = OFF

Segments

(Command-line equivalent = **/s**)

The Segments option adds a "Detailed map of segments" to the map file created with the Publics option (**/m**). The detailed list of segments contains the segment class, the segment name, the segment group, the segment module, and the segment ACBP information. If the same segment appears in more than one module, each module appears as a separate line.

The ACBP field encodes the A (alignment), C (combination), and B (big) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields: A, C, and B. The ACBP value in the map is printed in hexadecimal. The following field values must be ORed together to arrive at the ACBP value printed.

Field	Value	Description
A (alignment)	00	An absolute segment
	20	A byte-aligned segment
	40	A word-aligned segment
	60	A paragraph-aligned segment
	80	A page-aligned segment
	A0	An unnamed absolute portion of storage
C (combination)	00	Cannot be combined
	08	A public combining segment
B (big)	00	Segment less than 64K
	02	Segment exactly 64K

With the Segments options enabled, public symbols with no references are flagged `idle`. An idle symbol is a publicly defined symbol in a module that was not referenced by an EXTDEF record or by any other module included in the link. For example, this fragment from the public symbol section of a map file indicates that symbols `Symbol1` and `Symbol3` are not referenced by the image being linked:

```
0002:00000874      Idle      Symbol1
0002:00000CE4                Symbol2
0002:000000E7      Idle      Symbol3
```

Publics

(Command-line equivalent = **/m**)

This option causes the linker to produce a map file that contains an overview of the application segments and two listings of the public symbols. The segments listing has a line for each segment, showing the segment starting address, segment length, segment name, and the segment class. The public symbols are broken down into two lists, the first showing the symbols in sorted alphabetically, the second showing the symbols in increasing address order. Symbols with absolute addresses are tagged `Abs`.

A list of public symbols is useful when debugging: many debuggers use public symbols, which lets you refer to symbolic addresses while debugging.

- For more information, see Linker|Map file.

Print mangled names in map file (Options|Project|Linker|Map File)

(Command-line equivalent = **/M**)

Prints the mangled C++ identifiers in the map file, not the full name. This can help you identify how names are mangled (mangled names are needed as input by some utilities).

Default = OFF

Include source line numbers (Options|Project|Linker|Map File)

(Command-line equivalent: **/l**, 16-bit only)

When the Include Source Line Numbers option is on, the linker includes source line numbers in the object map files.

For this option to work, linked .OBJ files must be compiled with debug information using **-v**.

When Include Source Line Numbers is on, make sure you turn Jump Optimizations off in the Optimization|16 bit Specific options page, otherwise the compiler might group together common code from multiple lines of source text during jump optimization, or it might reorder lines (which makes line-number tracking difficult).

Default = OFF

Linker | 16-bit linker (Options|Project|Linker|16-bit Linker)

16-bit Linker options tell the linker how to link 16-bit programs.

The options are

Initialize segments

Inhibit optimizing far call to near

Enable 32-bit processing

Segment alignment

Linker goodies

Discard nonresident name table

Transfer resident names to nonresident names table

Initialize segments (Options|Project|Linker|16-bit Linker)

`(Command-line equivalent = /i, 16-bit only)`

When the Initialize Segments option is on, the linker initializes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This is normally not needed and will increase the size of your .EXE files.

Default = OFF

Inhibit optimizing far call to near (Options|Project|Linker|16-bit Linker)

(Command-line equivalent = **/f**, 16-bit only)

When the linker patches two code segments together, and far calls are made from one to the other, the linker will optimize the code by converting the far calls to near calls. When Inhibit Optimizing Far Call To Near is enabled, this optimization does not occur.

You might want to enable this option when you experience run-time crashes that appear to be related to corrupt virtual tables. Because virtual tables reside in the code segment, their contents can sometimes be interpreted by the linker as one of these far calls.

Default = OFF

Enable 32-bit processing (Options|Project|Linker|16-bit Linker)

(Command-line equivalent = **/3**, 16-bit only)

The Enable 32-bit processing option lets you link 32-bit DOS object modules produced by TASM or a compatible assembler. This option increases the memory requirements for TLINK and slows down linking.

Default = OFF

Segment alignment (Options|Project|Linker|16-bit Linker)

(Command-line equivalent = **/A:*dd***, 16-bit only)

Use the Segment Alignment input box to change the current byte value on which to align segments. The operating system seeks pages for loading based on this alignment value. You can enter numbers in the range of 2 to 65,535.

- The alignment factor is automatically rounded up to the nearest power of two. For example, if you enter 650, it is rounded up to 1,024 (this is different from the 32-bit Segment Alignment option).

For efficiency, you should use the smallest value that still allows for correct segment offsets in the segment table.

Default = 512

Discard nonresident name table (Options|Project|Linker|16-bit Linker)

`(Command-line equivalent = /Gn, 16-bit only)`

When the Discard Nonresident Name Table option is enabled, the linker does not emit the nonresident name table. The resultant image will contain only the module description in the nonresident names table.

See Transfer resident names to nonresident names table for usage details.

Default = OFF

Transfer resident names to nonresident names table (Options|Project|Linker|16-bit Linker)

(Command-line equivalent = **/Gr**, 16-bit only)

This option causes the linker to copy all names in the resident names table which have not been specified as RESIDENTNAME in the .DEF file to the nonresident names table. The resultant image contains only the module name and the symbol names of the exported symbols that were specified as RESIDENTNAME in the .DEF file.

When you use this option, you *must* also specify the WEP entry point as a RESIDENTNAME in the EXPORTS section of the .DEF file (Windows obtains the WEP entry point for this symbol by looking it up in the resident names table).

- When building .DLLs that contain many exports, it's possible to exceed the 64K header file limitation. Because the .DLL contains the resident names table in its header, moving the exports out of the header using the **/Gr** option usually remedies this problem. The **/Gr** option causes the linker to transfer the names in the resident names table to the nonresident names table. Names in the nonresident names table are then assigned ordinal numbers, which your .EXE file uses when referencing the entry points in the .DLL.

There are two ways to create input files for the linker:

- Run IMPLIB on the .DLL to create an import library for linking purposes.
- Run IMPDEF in the .DLL to create a .DEF file for linking purposes.

Once the import library or .DEF file has been created, there is no need to keep the names in either the resident or the nonresident names tables. Relinking the .DLL and specifying *both* the Transfer Resident Names to Nonresident Names Table (**/Gr**) and Discard Nonresident Name Table (**/Gn**) options causes the linker to build a .DLL with an "empty" names table. Not only does this post-processing avoid the problem of exceeding the header limitation, but it also creates a .DLL that loads faster (because it's smaller) and runs faster (because references to entry points are by ordinal number instead of by name).

To summarize this process, you must

1. Enable the **/Gr** switch to transfer the names in the resident names table to the nonresident names table. This also assigns ordinal numbers to the names. However; before doing so, make sure you have included a .DEF file with the following export definition in the EXPORTS section:

   ```
   EXPORTS
         WEP @1 RESIDENTNAME
   ```

2. Build the .DLL.
3. Run IMPLIB or IMPDEF on the new .DLL file.
4. Enable the **/Gn** switch (along with the already enabled **/Gr** switch).
5. Relink the .DLL.

To see an example of this process, refer to the makefile that builds the ObjectWindows example programs.

Default = OFF

Linker | 16-bit optimizations (Options|Project|Linker|16-bit Optimizations)

The 16-bit Optimizations control how the linker optimizes 16-bit .EXE programs. In most cases the final executable file size is reduced, which results in a faster load time.

Whenever you use one or more of these options, the linker reorders the .EXE segments as follows:

PRELOAD segments

PRELOAD resources

LOAD ON CALL segments

LOAD ON CALL resources

The options are:

Chain fixes

Iterate data

Minimize segment alignment

Minimize resource alignment

- These options work only with 16-bit Windows and DPMI programs.

Chain fixes (Options|Project|Linker|16-bit Optimizations)

`(Command-line equivalent = /Oc, 16-bit only)`

Chain fixes removes duplicate and/or unnecessary fixup data from the .EXE file. This is done by emitting only one fixup record for each unique internal fixup and ”remembering” the duplicate fixes by creating a linked list of the internal fixup locations within the .EXE data segment. When the loader loads the .EXE, it applies the fixup specified in the fixup record to each of the locations specified in the linked list. Specifying this optimization also causes trailing zeros in data segments to be eliminated. This usually results in a significantly smaller .EXE file, which loads faster.

Default = OFF

Iterate data (Options|Project|Linker|16-bit Optimizations)

`(Command-line equivalent = /Oi, 16-bit only)`

This option scans data segments for patterns of data (for example, a block with 128 bytes filled with “0”). Instead of emitting the data, TLINK emits a “description” of the block of data which matches the pattern (for example, a 5-byte descriptor specifying a 128 bytes of 0). Specifying this optimization also causes trailing zeros in data segments to be eliminated. This usually results in a significantly smaller .EXE file, which loads faster.

Default = OFF

Minimize segment alignment (Options|Project|Linker|16-bit Optimizations)

(Command-line equivalent = **/Oa**, 16-bit only)

This optimization switch determines the minimum segment alignment value by examining the size of the .EXE file. An .EXE that has a size of 1 byte to 64K bytes results in an alignment value of 1; if the .EXE file size is 64K+1 bytes to 128K bytes, the alignment value is 2; and so on.

While this optimization results in a smaller .EXE file, the .EXE might load slower because the newly calculated alignment value may cause the segments to cross physical disk sector boundaries more often. Unless you have also specified the Segment Alignment (**/A**) linker option, the linker initially generates an .EXE using the default alignment value of 512.

- This option setting overrides whatever alignment value the linker might have used to initially generate the .EXE file.

Default = OFF

Minimize resource alignment (Options|Project|Linker|16-bit Optimizations)

(Command-line equivalent = **/Or**, 16-bit only)

This optimization switch is the same as the Minimize segment alignment switch (**/Oa**), except that it applies to resource alignment values instead of segment alignment values.

Default = OFF

Linker | 32-bit linker (Options|Project|Linker|32-bit Linker)

32-bit Linker options tell the linker how to link 32-bit programs.

The options are

Image is based
Allow import by ordinal
Verbose
Maximum linker errors
Use incremental linker

Image is based (Options|Project|Linker|32-bit Linker)

The Image is Based option affects whether an application has an image base address. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the number specified in the Image Base Address input box. Using this option can greatly reduce the size of your final application module; however, it is not recommended for use when producing a DLL.

Default = OFF

Allow import by ordinal (Options|Project|Linker|32-bit Linker)

`(Command-line equivalent = /o, 32-bit only)`

This option lets you import by ordinal value instead of by the import name. When you specify this option, the linker emits only the ordinal numbers (and not the import names) to the resident or nonresident name table for those imports that have an ordinal number specified. If you do not specify this option, the linker ignores all ordinal numbers contained in import libraries or the .DEF file, and emits the import names to the resident and nonresident tables.

- This option is different than the 16-bit **/o** (overlays) option.

Verbose (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/r**, 32-bit only)

This option causes the linker to emit messages that indicate what part of the link cycle is currently being executed by the linker. With this option turned on, the linker emits some or all of the following messages:

```
Starting pass 1
Generating map file
Starting pass 2
Reading resource files
Linking resources
```

Maximum linker errors (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/E*nn***)

Specifies maximum errors the linker reports before terminating. **/E0** (default) reports an infinite number of errors (that is, as many as possible).

Use incremental linker (Options|Project|Linker|32-bit Linker)

Uses the incremental linker.

When on, the first link of the executable file takes about the same amount of time as without the incremental linker. On subsequent links, when you make small changes to your source code, the link increases in speed. With the incremental linker in use, the link is usually less than 2 seconds, even for multiple megabyte images.

Linker | 32-bit image (Options|Project|Linker|32-bit Linker)

32-bit Image options control the application image.

The options are

Image base address (in hexadecimal)
File alignment (in hexadecimal)
Object alignment (in hexadecimal)
Reserved stack size (in hexadecimal)
Committed stack size (in hexadecimal)
Reserved heap size (in hexadecimal)
Committed heap size (in hexadecimal)

Image base address (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/B:*xxxx***, 32-bit only)

The Image Base Address option specifies an image base address for an application, and is used in conjunction with the Image is based option. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the hexadecimal number specified. All successive objects are aligned on 64K linear address boundaries. This option makes applications smaller on disk and improves both load-time and run-time performance (the operating system no longer has to apply internal fixes).

The command-line version of this option (**/B:*xxxx***) accepts either decimal or hexadecimal numbers as the image base address.

▪ It is not recommended that you enable this option when producing a DLL. In addition, do not use the default setting of 0x400000 if you intend to run your application of Win32s systems.

Default = 0x400000 (recommended for true Win32 system applications)

File alignment (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/Af:*xxxx***, 32-bit only)

The File Alignment option specifies page alignment for code and data within the executable file. The linker uses the file alignment value when it writes the various objects and sections (such as code and data) to the file. For example, if you use the default value of 0x200, the linker stores the section of the image on 512-byte boundaries within the executable file.

When using this option, you must specify a file alignment value that is a power of 2, with the smallest value being 16.

- The old style of this option (**/A:*dd***) is still supported for backward compatibility. With this option, the decimal number ***dd*** is multiplied by the power of 2 to calculate the file alignment value.

The command-line version of this option (**/Af:*xxxx***) accepts either decimal or hexadecimal numbers as the file alignment value.

Default = 512 (0x200)

Object alignment (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/Ao:*xxxx***, 32-bit only)

The linker uses the object alignment value to determine the virtual addresses of the various objects and sections (such as code and data) in your application. For example, if you specify an object alignment value of 8192, the linker aligns the virtual addresses of the sections in the image on 8192-byte (0x2000) boundaries.

When using this option, you must specify an object alignment value that is a power of 2, with the smallest value being 4096 (the default).

The command-line version of this option (**/Ao:*xxxx***) accepts either decimal or hexadecimal numbers as the object alignment value.

Default = 4096 (0x1000)

Reserved stack size (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/S:*xxxx***, 32-bit only)

Specifies the size of the reserved stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000).

- Specifying the reserved stack size here overrides any STACKSIZE setting in a module definition file.

The command-line version of this option (**/S:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Committed stack size (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/Sc:*xxxx***, 32-bit only)

Specifies the size of the committed stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000) and any value specified must be equal to or less than the Reserved Stack Size setting (**/S**).

- Specifying the committed stack size here overrides any STACKSIZE setting in a module definition file.

The command-line version of this option (**/Sc:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 8K (0x2000)

Reserved heap size (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/H:*xxxx***, 32-bit only)

Specifies the size of the reserved heap in hexadecimal. The minimum allowable value for this field is 0.

- Specifying the reserved heap size here overrides any HEAPSIZE setting in a module definition file.

The command-line version of this option (**/H:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Committed heap size (in hexadecimal) (Options|Project|Linker|32-bit Linker)

(Command-line equivalent = **/Hc:*xxxx***, 32-bit only)

Specifies the size of the committed heap in hexadecimal. The minimum allowable value for this field is 0 and any value specified must be equal to or less than the Reserved Heap Size setting (**/H**).

- Specifying the committed heap size here overrides any HEAPSIZE setting in a module definition file.

The command-line version of this option (**/Hc:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 4K (0x1000)

Linker | Warnings (Options|Project|Linker|Warnings)

Warnings options enable or disable the display of Linker warnings.

The options are

Warn duplicate symbol in .LIB

"No Stack" warning

32-bit warnings

No entry point

Duplicate symbol

No def file

Import does not match previous definition

Extern not qualified with _import

Using based linking in DLL

Self-relative fixup overflowed

.EXE module built with a .DLL extension

Multiple stack segments found

Warn duplicate symbol in .LIB **(Options|Project|Linker|Warnings)**

See Also

(Command-line equivalent = **/d** 16-bit, **/wdpl** 32-bit)

When the Warn Duplicate Symbols option is on, the linker warns you if a symbol appears in more than one object or library files.

If the symbol must be included in the program, the linker uses the symbol definition from the first file it encounters with the symbol definition.

Use the TLINK32 command-line option **/w-dpl** to turn this warning off.

Default = OFF

"No stack" warning (Options|Project|Linker|Warnings)

(Command-line equivalent = **/k** 16-bit, **/wstk** 32-bit)

This option lets you control whether or not the linker emits the "No stack" warning. The warning is generated if no stack segment is defined in any of the object files or in any of the libraries included in the link. Except for .DLLs, this indicates an error. If a Borland C++ program produces this error, make sure you are using the correct C0*x* startup object file.

Use the TLINK32 command-line option **/w-stk** to turn this warning off.

Default = OFF

Make options (Options|Project|Make)

Make options control the conditions under which the building of a project stops and how the project manager uses autodependency information.

The options are

Break make on

Autodependencies

New node path

Make | Break make on (Options|Project|Make)

The Make|Break Make On options specify the error condition that stops the making of a project.

Warnings

`(Command-line equivalent = -w!)`

This option stops a make if the compiler encounters warnings.

When this compiler option is enabled, the compiler terminates the compile and returns a non-zero error code if a warning is encountered; an .OBJ file is not created.

Errors

This option stops a make when the compiler encounters errors.

Fatal errors

This option tells the Project Manager to generate a list of errors and warnings for all files and all targets in the project. The Project Manager will go on to link if no errors occur.

Default = Errors

Make | Autodependencies (Options|Project|Make)

When the Make|Autodependencies option is selected, the Project Manager automatically checks dependencies for every target that has a corresponding source file in the project list.

None

When None is selected, no autodependency checking is performed.

Use

When Use is selected, autodependency checking is performed by reading the autodependency information out of the .OBJ files.

Cache

When Cache is selected, autodependency information is stored in memory to make dependency checking faster. This option speeds up compilation, but autodependency information will not display in the project tree.

Cache and display

When Cache and Display is selected, the Project Manager stores the autodependency information in the project file. Once the autodependency information is generated (after a compile) the information is displayed in the project tree. This makes dependency checking faster, but makes project files larger.

Make | New node path (Options|Project|Make)

Turn on the Absolute option if you want new nodes to have an absolute, instead of a relative, path.

Messages Options (Options|Project|Messages)

Messages options let you control the messages generated by the compiler. Compiler messages are indicators of potential trouble spots in your program. These messages can warn you of many problems that may be waiting to happen, such as variables and parameters that are declared but never used, type mismatches, and many others.

Setting a message option causes the compiler to generate the associated message or warning when the specific condition arises. Note that some of the messages are on by default.

The subtopics are

Portability

ANSI violations

Obsolete C++

Potential C++ errors

Inefficient C++ coding

Potential errors

Inefficient coding

General

Display Warnings

All

Selected

None

Stop after ... warnings

Stop after ... errors

Display warnings (Options|Project|Messages)

Use the Display Warnings options to choose which warnings are displayed.

All

(Command-line equivalent: **`-w`**)

Display all warning and error messages.

Default = OFF

Selected

(Command-line equivalent: -***`waaa`***)

Choose which warnings are displayed. Using `pragma warn` in your source code overrides messages options set either at the command line or in the IDE.

To disable a message from the command line, use the command-line option **`-w-`*****`aaa`***, where ***aaa*** is the 3-letter message identifier used by the command-line option.

Default = ON

None

Suppresses the display of warning messages. Errors are still displayed.

Default = OFF

Stop after ... warnings (Options|Project|Messages)

(Command-line equivalent: **-g*n***)

Warnings: Stop After causes compilation to stop after the specified number of warnings has been detected. You can enter any number from 0 to 255.

Entering 0 causes compilation to continue until either the end of the file or the error limit set in Errors: Stop After has been reached, whichever comes first.

Default = 100

Stop after ... errors (Options|Project|Messages)

(Command-line equivalent: `-j`*n*)

Errors: Stop After causes compilation to stop after the specified number of errors has been detected. You can enter any number from 0 to 255.

Entering 0 causes compilation to continue until the end of the file.

Default = 25

Messages | Portability (Options|Project|Messages|Portability)

Compiler Messages|Portability options enable or disable individual warning messages about statements that might not operate correctly in all computer environments.

Option	Command-line equivalent	Default
Non-portable pointer conversion	**-w-rpt**	ON
Non-portable pointer comparison	**-w-cpt**	ON
Constant out of range in comparison	**-w-rng**	ON
Constant is long	**-wcln**	OFF
Conversion may lose significant digits	**-wsig**	OFF
Mixing pointers to signed and unsigned char	**-wucp**	OFF

Messages | ANSI Violations (Options|Project|Messages|ANSI Violations)

Compiler Messages|ANSI Violations options enable or disable individual warning messages about statements that violate the ANSI standard for the C language.

Option	Command-line equivalent	Default
Void functions may not return a value	**-w-voi**	ON
Both return and return of a value used	**-w-ret**	ON
Suspicious pointer conversion	**-w-sus**	ON
Undefined structure 'ident'	**-wstu**	OFF
Redefinition of 'ident' is not identical	**-w-dup**	ON
Hexadecimal value more than three digits	**-w-big**	ON
Bit fields must be signed or unsigned int	**-wbbf**	OFF
'ident' declared as both external and static	**-w-ext**	ON
Declare 'ident' prior to use in prototype	**-w-dpu**	ON
Division by zero	**-w-zdi**	ON
Initializing 'ident' with 'ident'	**-w-bei**	ON
Initialization is only partially bracketed	**-wpin**	OFF
Non-ANSI keyword used	**-wnak**	OFF

Messages | Obsolete C++ (Options|Project|Messages|Obsolete C++)

Compiler Messages|Obsolete C++ options choose which specific obsolete items or incorrect syntax C++ warnings to display.

Option	Command-line equivalent	Default
Base initialization without class name is obsolete	**-w-obi**	ON
This style of function definition is obsolete	**-w-ofp**	ON
Overloaded prefix operator used as a postfix operator	**-w-pre**	ON

Messages | Potential C++ Errors (Options|Project|Messages|Potential C++ Errors)

Compiler Messages|Potential C++ Errors options enable or disable individual warning messages about statements that violate C++ language implementation.

Option	Command-line equivalent	Default
Constant member 'ident' is not initialized	`-w-nci`	ON
Assigning 'type' to 'enumeration'	`-w-eas`	ON
'function' hides virtual function 'function2'	`-w-hid`	ON
Non-const function <function> called for const object	`-w-ncf`	ON
Base class 'ident' inaccessible because also in 'ident'	`-w-ibc`	ON
Array size for 'delete' ignored	`-w-dsz`	ON
Use qualified name to access nested type 'ident'	`-w-nst`	ON
Handler for '<type1>' Hidden by Previous Handler for '<type2>'	`-w-hch`	ON
Conversion to 'type' will fail for virtual base members	`-w-mpc`	ON
Maximum precision used for member pointer type <type>	`-w-mpd`	ON
Use '> >' for nested templates instead of '>>'	`-w-ntd`	ON
Non-volatile function <function> called for volatile object	`-w-nvf`	ON

Messages | Inefficient C++ coding (Options|Project|Messages|Inefficient Coding)

Compiler Messages|Inefficient C++ Coding options enable or disable individual warning messages about inefficient C++ coding.

Option	Command-line equivalent	Default
Functions containing 'ident' not expanded inline	**-w-inl**	ON
Temporary used to initialize 'ident'	**-w-lin**	ON
Temporary used for parameter 'ident'	**-w-lvc**	ON

Messages | Potential errors (Options|Project|Messages|Potential Errors)

Compiler Messages|Potential Errors options enable or disable individual warning messages about potential coding errors.

Option	Command-line equivalent	Default
Possibly incorrect assignment	**-w-pia**	ON
Possible use of 'ident' before definition	**-wdef**	OFF
No declaration for function 'ident'	**-wnod**	OFF
Call to function with no prototype	**-w-pro**	ON
Function should return a value	**-w-rvl**	ON
Ambiguous operators need parentheses	**-wamb**	OFF
Condition is always (true/false)	**-w-ccc**	ON
Continuation character \ found in //	**-w-com**	ON

Messages | Inefficient coding (Options|Project|Messages|Inefficient Coding)

Compiler Messages|Inefficient Coding options are used to enable or disable individual warning messages about inefficient coding.

Option	Command-line equivalent	Default
'ident' assigned a value which is never used	**-w-aus**	ON
Parameter 'ident' is never used	**-w-par**	ON
'ident' declared but never used	**-wuse**	OFF
Structure passed by value	**-wstv**	OFF
Unreachable code	**-w-rch**	ON
Code has no effect	**-w-eff**	ON

- The warnings `Unreachable Code` and `Code Has No Effect` can indicate serious coding problems. If the compiler generates these warnings, be sure to examine the lines of code that cause these warnings.

Messages | General (Options|Project|Messages|General)

Compiler Messages|General options enable or disable a few general warning messages.

Option	Command-line equivalent	Default
Unknown assembler instruction	`-wasm`	OFF
Ill-formed pragma	`-w-ill`	ON
Array variable 'ident' is near	`-w-ias`	ON
Superfluous & with function	`-wamp`	OFF
'ident' is obsolete	`-w-obs`	ON
Cannot create precompiled header	`-w-pch`	ON
User-defined warnings	`-w-msg`	ON

User-defined warnings (Options|Project|Messages|General)

(Command-line equivalent: **-wmsg**)

The User-defined warnings option allows user-defined messages to appear in the IDE's Message window. User-defined messages are introduced with the #pragma message compiler syntax.

- In addition to messages that you introduce with the #pragma message compiler syntax, User-defined warnings displays warnings introduced by third-party libraries. Remember, if you need Help on a third-party warning, please contact the vendor of the header file that issued the warning.

Default = ON

Optimization options (Options|Project|Optimizations)

See also

Optimization options are the software equivalent of performance tuning. There are two general types of compiler optimizations:

- Those that make your code smaller
- Those that make your code faster

Although you can compile with optimizations at any point in your product development cycle, be aware when debugging that some assembly instructions might be "optimized away" by certain compiler optimizations.

General settings

The main Optimizations page in the Project Options dialog box contains four radio buttons that let you select the overall type of optimizations you want to use. Because of the complexities of setting compiler optimizations, it is recommended that you use either the Optimize for Size or the Optimize for Speed radio buttons. The general optimization settings are:

Disable all optimizations

Use selected optimizations

Optimize for size

Optimize for speed

- If you import an .IDE file from Borland C++ version 4.5x that had Disable all optimizations selected, the setting is changed to Use selected optimizations. If desired, you can reset the selection to its original setting.

Optimization subtopics

If you have a special need for certain compiler optimizations, specifically set the optimizations you need using the settings in the Optimization subtopics. The subtopic pages are

16 and 32-bit

16-bit specific

32-bit specific

General optimization settings (Options|Project|Optimizations)

Disable all optimizations
(Command-line equivalent: **-Od**)

Disables all optimization settings, including ones which you may have specifically set and those which would normally be performed as part of the speed/size tradeoff.

Because this disables code compaction (tail merging) and cross-jump optimizations, using this option can keep the debugger from jumping around or returning from a function without warning, which makes stepping through code easier to follow.

- You can override this setting using the predefined Style Sheets in the Project Manager.

Use selected optimizations
Does not set any optimization by default, but lets you set the specific optimization options you need through the settings contained in the Optimization subtopics. The subtopic pages are

16 and 32-bit

16-bit specific

32-bit specific

- Configuring your own optimization settings should be reserved for expert users only.

Optimize for size
(Command-line equivalents: **-O1**)

This radio button sets an aggregate of optimization options that tells the compiler to optimize your code for size. For example, the compiler scans the generated code for duplicate sequences. When such sequences warrant, the optimizer replaces one sequence of code with a jump to the other and eliminates the first piece of code. This occurs most often with **switch** statements. The compiler optimizes for size by choosing the smallest code sequence possible.

This option (**-O1**) sets the following optimizations:

- Jump optimizations (**-O**)
- Dead code elimination (**-Ob**)
- Duplicate expressions (**-Oc**)
- Register allocation and live range analysis (**-Oe**)
- Loop optimizations (**-Ol**)
- Instruction scheduling (**-OS**)
- Register load suppression (**-Z**)
- The compiler options **-Ot** and **-G** are supported for backward compatibility only, and are equivalent to the **-O1** compiler option.

Optimize for speed
(Command-line equivalent: **-O2**)

This radio button sets an aggregate of optimization options that tells the compiler to optimize your code for speed. This switch (**-O2**) sets the following optimizations:

- Dead code elimination (**-Ob**)
- Register allocation and live range analysis (**-Oe**)
- Duplicate expression within functions (**-Og**)
- Intrinsic functions (**-Oi**)
- Loop optimizations (**-Ol**)
- Code motion (**-Om**)
- Copy propagation (**-Op**)
- Instruction scheduling (**-OS**)
- Induction variables (**-Ov**)
- Register load suppression (**-Z**)

If you are creating Windows applications, you'll probably want to optimize for speed.

- The compiler options **`-Os`** and **`-G-`** are supported for backward compatibility only, and are equivalent to the **`-O2`** compiler option. The **`-Ox`** option is also supported for backward compatibility and for compatibility with Microsoft make files.

Optimizations | 16 and 32-bit (Options|Project|Optimizations|16 and 32-bit)

The 16- and 32-bit compiler options specify optimization settings for all compilations. The options are

Common subexpressions
No optimization
Optimize locally
Optimize globally

General
Inline intrinsic functions
Induction variables

Common subexpression (Options|Project|Optimizations|16 and 32-bit)

The Common Subexpressions options tell the compiler how to find and eliminate duplicate expressions in your code.

No optimization

When the No Optimization option is on, the compiler does not eliminate common subexpressions. This is the default behavior of the command-line compilers.

Optimize locally

`(Command-line equivalent: -Oc)`

When the Optimize Locally option is on, the compiler eliminates common subexpressions within groups of statements unbroken by jumps (basic blocks).

Optimize globally

`(Command-line equivalent: -Og)`

When you set this option, the compiler eliminates common subexpressions within an entire function. This option globally eliminates duplicate expressions within the target scope and stores the calculated value of those expressions once (instead of recalculating the expression).

Although this optimization could theoretically reduce code size, it optimizes for speed and rarely results in size reductions. Use this option if you prefer to reuse expressions rather than create explicit stack locations for them.

Inline intrinsic functions (Options|Project|Optimizations|16 and 32-bit)

(Command-line equivalent: **-Oi**)

When the Inline Intrinsic Functions option is on, the compiler generates the code for common memory functions like strcpy() within your function's scope. This eliminates the need for a function call. The resulting code executes faster, but it is larger.

The following functions are inlined with this option:

alloca	*fabs*	*memchr*	*memcmp*
memcpy	*memset*	*rotl*	*rotr*
stpcpy	*strcat*	*strchr*	*strcmp*
strcpy	*strlen*	*strncat*	*strncmp*
strncpy	*strnset*	*strrchr*	

You can control the inlining of these functions with the pragma **intrinsic**. For example, `#pragma intrinsic strcpy` causes the compiler to generate inline code for all subsequent calls to **strcpy** in your function, and `#pragma intrinsic -strcpy` prevents the compiler from inlining **strcpy**. Using these pragmas in a file overrides any compiler option settings.

When inlining any intrinsic function, you must include a prototype for that function before you use it; the compiler creates a macro that renames the inlined function to a function that the compiler recognizes internally. In the previous example, the compiler would create a macro `#define strcpy _ _strcpy_ _`.

The compiler recognizes calls to functions with two leading and two trailing underscores and tries to match the prototype of that function against its own internally stored prototype. If you don't supply a prototype, or if the prototype you supply doesn't match the compiler's prototype, the compiler rejects the attempt to inline that function and generates an error.

Induction variables (Options|Project|Optimizations|16 and 32-bit)

(Command-line equivalent: **-Ov**)

When this option is enabled, the compiler creates induction variables and it performs strength reduction, which optimizes for loops speed.

Use this option when you're compiling for speed and your code contains loops. The optimizer uses induction to create new variables (induction variables) from expressions used in loops. The optimizer assures that the operations performed on these new variables are computationally less expensive (reduced in strength) than those used by the original variables.

Optimizations are common if you use array indexing inside loops, because a multiplication operation is required to calculate the position in the array that is indicated by the index. For example, the optimizer creates an induction variable out of the operation `v[i]` in the following code because the `v[i]` operation requires multiplication. This optimization also eliminates the need to preserve the value of `i`:

```
int v[10];
void f(int x, int y, int z)
{
  int i;
  for (i = 0; i < 10; i++)
     v[i] = x * y * z;
}
```

With Induction variables enabled, the code changes:

```
int v[10];
void f(int x, int y, int z)
{
  int i, *p;
  for (p = v; p < &v[9]; p++)
    *p = x * y * z;
}
```

Optimizations | 16-bit (Options|Project|Optimizations|16-bit)

The Optimizations|16-bit options pertain to 16-bit applications only. These options are

Jump optimizations
Loop optimization
Suppress redundant loads
Dead code elimination
Windows prolog/epilog
Global register allocation
Assume no pointer aliasing
Invariant code motion
Copy propagation

Jump optimization (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-O**)

When Jump Optimization option is on, the compiler reduces the code size by eliminating redundant jumps and reorganizing loops and switch statements.

When this option is enabled, the sequences of stepping in the debugger can be confusing because of the reordering and elimination of instructions. If you are debugging at the assembly level, you might want to disable this option.

Default = ON

Loop optimization (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Ol**)

When this option is enabled, loops are compacted into REP/STOSx instructions.

Loop optimization takes advantage of the string move instructions on the 80x86 processors by replacing the code for a loop with a string move instruction, making the code faster.

Depending on the complexity of the operands, the compacted loop code can also be smaller than the corresponding non-compacted loop.

Suppress redundant loads (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Z**)

When this option is enabled, the compiler suppresses the reloading of registers by remembering the contents of registers and reusing them as often as possible.

Exercise caution when using this option; the compiler cannot detect if a value has been modified indirectly by a pointer.

Dead code elimination (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Ob**)

When the Dead Code Elimination option is on, the compiler reveals variables that might not be needed. Because the optimizer must determine where variables are no longer used (live range analysis), you might also want to set Global Register Allocation (**-Oe**) when you use this option.

Windows prolog/epilog (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-OW**)

Use the Windows Prolog/Epilog option to suppress the **inc bp** / **dec bp** of an exported Windows far function prolog and epilog code.

If Debug Information in .OBJ Files (**-v**) is enabled, this option is disabled because some debugging tools (such as WinSpector and Turbo Debugger) need the **inc bp** / **dec bp** instructions to display stack-frame information.

Global register allocation (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Oe**)

When this option is enabled, global register allocation and variable live range analysis are enabled. This option should always be used when optimizing code because it increases the speed and decreases the size of your application.

Assume no pointer aliasing (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Oa**)

When the Assume No Pointer Aliasing option is on, the compiler assumes that pointer expressions are not aliased in common subexpression evaluation.

Assume No Pointer Aliasing affects the way the optimizer performs common subexpression elimination and copy propagation by letting the optimizer maintain copy propagation information across function calls and by letting the optimizer maintain common subexpression information across some stores. Without this option the optimizer must discard information about copies and subexpressions. Pointer aliasing might create bugs that are hard to spot, so it is only applied when you enable this option.

Assume No Pointer Aliasing controls how the optimizer treats expressions that contain pointers. When compiling with global or local common subexpressions and Assume No Pointer Aliasing is enabled, the optimizer recognizes `*p * x` as a common subexpression in function *func1*.

```
int g, y;
int func1(int *p)
{
  int x=5;
  y = *p * x;
  g = 3;
  return (*p * x);
}
void func2(void)
{
  g=2;
  func1(&g);  // This is incorrect--the assignment g = 3
              // invalidates the expression *p * x
}
```

Invariant code motion (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Om**)

When this option is enabled, invariant code is moved out of loops and your code is optimized for speed. The optimizer uses information about all the expressions in the function (gathered during common subexpression elimination) to find expressions whose values do not change inside a loop.

To prevent the calculation from being done many times inside the loop, the optimizer moves the code outside the loop so that it is calculated only once. The optimizer then reuses the calculated value inside the loop.

You should use loop-invariant code motion whenever you are compiling for speed and have used global common subexpressions, because moving code out of loops can result in enormous speed gains. For example, in the following code, `x * y * z` is evaluated in every iteration of the loop:

```
int v[10];
void f(int x, int y, int z)
{
  int i;
  for (i = 0; i < 10; i++)
    v[i] = x * y * z;
}
```

The optimizer rewrites the code:

```
int v[10];
void f(int x, int y, int z)
{
  int i,t1;
  t1 = x * y * z;
  for (i = 0; i < 10; i++)
    v[i] = t1;
}
```

Copy propagation (Options|Project|Optimizations|16-bit)

(Command-line equivalent: **-Op**)

When this option is enabled; copies of constants, variables, and expressions are propagated whenever possible.

Copy propagation is primarily speed optimization, but it never increases the size of your code. Like loop-invariant code motion, copy propagation relies on the analysis performed during common subexpression elimination. Copy propagation means that the optimizer remembers the values assigned to expressions and uses those values instead of loading the value of the assigned expressions. With this, copies of constants, expressions, and variables can be propagated.

Optimizations | 32-bit (Options|Project|Optimizations|32-bit)

Use the Optimizations|32-bit options to specify options specific to the Pentium processor. The options are:

Pentium instruction scheduling
(Command-line equivalent: **-OS**)

When enabled, this switch rearranges instructions to minimize delays that can be caused by Address Generation Interlocks (AGI) which occur on the i486 and Pentium processors. This option also optimizes the code so that it takes advantage of the Pentium parallel pipelines. Best results for Pentium systems are obtained when you use this switch in conjunction with the 32-bit Compiler|Pentium option in the Project Options dialog box (-5).

- Scheduled code is more difficult to debug at the source level because instructions from a particular source line may be mixed with instructions from other source lines. Stepping through the source code is still possible, although the execution point might make unexpected jumps between source lines as you step. Also, setting a breakpoint on a source line may result in several breakpoints being set in the code. This is especially important to note when inspecting variables, since a variable may be undefined even though the execution point is positioned after the variable assignment.

Stepping through the following function when this switch is enabled demonstrates the stepping behavior:

```
int v[10];
void f(int i, int j)
{
  int a,b;

  a = v[i+j];
  b = v[i-j];
  v[i] = a + b;
  v[j] = a - b;
}
```

Execution starts by computing the index `i-j` in the assignment to `b` (note that `a` is still undefined although the execution point is positioned after the assignment to `a`). The index `i+j` is computed, `v[i-j]` is assigned to `b`, and `v[i+j]` is assigned to `a`. If a breakpoint is set on the assignment to `b`, execution will stop twice: once when computing the index and again when performing the assignment.

Default = OFF (-O-S)

Resources options (Options|Project|Resources)

The Resources options let you set several options for how resources are compiled and bound to your application by the integrated resource compiler and resource linker.

The options are

16-bit Resources

32-bit Resources

Tip: You cannot mix 16-bit and 32-bit files in a resource project. To add resources from existing 32-bit .RES or .EXE files to a 16-bit resource project, save the files as .RC files and add the .RC files to your project; .RC files are not version specific.

16-bit Resources (Options|Project|Resources)

The 16-bit Resource option applies to resources targeted for 16-bit applications.

Pack fastload area

The Pack Fastload Area option optimizes the executable file by packing all PRELOAD segments and resources in a contiguous area in the. EXE file.

When Pack Fastload Area is enabled, the resource linker writes all data segments, all nondiscardable code segments, and the entry-point code segment to a contiguous area in the executable file. This makes your executable files load faster at runtime.

If you uncheck Pack Fastload Area, no such optimizations will be done.

Default = OFF

32-bit Resources (Options|Project|Resources)

The 32-bit resource options apply to resources targeted for 32-bit applications.

Language

Use the Language list boxes to select the base and sub-languages to be used for the resources in the your project.

Multiple resources with the same ID may be bound to a single .EXE or .DLL if they each have different major and/or minor languages specified. The language currently specified by Windows will determine which resource identified by the ID will actually be loaded.

The language you choose must match the language that Windows NT systems are set to. If not, your resources might not be correctly displayed (this is especially true for menu resources).

Major Use the Major list box to select the base (or primary) language to be used for the resources.

Minor Use the Minor list box to select the sub-language (or dialect) to be used for the resources.

For example, you might define your resource to have English as the base language and U.S. as the sub-language.

- These options set the default languages for the Resource compiler. You can override these default settings by embedding language statements in your .RC file.

Default = Neutral

Browser Options (Options|Environment)

The Options|Environment|Browser options determine which symbols can be viewed using the Browser and whether one or more Browser windows can be open at the same time.

The options are:

Visible Symbols

Browser Window Behavior

Visible Symbols (Options|Environment|Browser)

Use the Environment|Browser|Visible Symbols options to specify the types of symbols you want to have visible in the Browser. These options define the initial display settings when you open a Browser view that has Browser Filters and Letter Symbols options.

You can change these options for each view by using the controls at the bottom of each of the open Browser views.

The types of symbols you can view are:

Constants	Display constants and defined values.
Types	Display data types.
Functions	Display functions.
Debuggable	Display 'debuggable' objects.
Variables	Display local and global variables.
Inherited	Display inherited classes, data members, and member functions.
Virtual	Display virtual classes, data members, and member functions.

Browser Window Behavior (Options|Environment|Browser)

The Environment|Browser|Browser Window Behavior options control how many Browser views you can have open at one time. These options specify the default setting for the start of the first Browser session. You can also set these options within a Browser view by choosing Toggle Window Mode on the Browser SpeedMenu.

Mode	Description
Single Window	Lets you open only one Browser view at a time. (It also enables the Return to Previous View command on the Browser SpeedMenu.) The next time you select a Browser action, the Browser replaces the view you currently have open.
Multiple Window	Lets you open multiple Browser views at the same time. Each time you select a new Browser action, an additional Browser view window opens.

Editor options (Options|Environment|Editor)

The Options|Environment|Editor options let you specify the appearance and behavior of the Borland C++ IDE editor.

The options are:

Editor SpeedSettings

The subtopics are:

Options

File

Display

Editor SpeedSettings (Options|Environment|Editor)

The Editor SpeedSettings in the Editor options let you set several editor-related options at one time. These options are set to emulate a specific type of editor. The display box in the lower right corner of the dialog shows which options are set by these SpeedSettings and the values are for each setting.

The available Editor SpeedSettings are:

Default keymapping
Uses the default key assignments for the editor, menus, and other IDE windows.

IDE classic
Uses the traditional Borland key assignments for the editor, menus, and other IDE windows.

BRIEF emulation
Uses Brief key assignments for the editor, menus, and other IDE windows.

Epsilon emulation
Uses Epsilon key assignments for the editor, menus, and other IDE windows.

- When you select an Editor SpeedSetting, a partial listing of the default behavior is displayed. These settings may not accurately indicate the active settings in the IDE if you changed individual settings on the OptionsFileDisplay pages:

  Group undo
  Cursor beyond EOF
  Keep Trailing blanks
  BRIEF RegEx
  Persistent blocks
  Overwrite blocks
  Keymap
  BRIEF regular expressions

Editor | Options (Options|Environment|Editor|Options)

The Editor|Options let you tailor the behavior of the IDE editor.

The Editor options are:

Auto indent mode	Cursor beyond EOF
Insert mode	Undo after save
Use tab character	Keep trailing blanks
Optimal fill	BRIEF regular expressions
Backspace unindents	Overwrite blocks
Cursor through tabs	Persistent blocks
Group undo	Double Click line
Cursor beyond EOL	
Block Indent	Undo Limit
Tab Stops	

Auto indent mode (Options|Environment|Editor|Options)

Positions the cursor under the first nonblank character in the preceding nonblank line when you press Enter in an Edit window.

This feature can help you keep your program code more readable.

You can use Optimal Fill in conjunction with Auto Indent Mode to automatically insert the smallest number of tabs and spaces to indent your code.

Insert mode (Options|Environment|Editor|Options)

Inserts the text you type and pushes existing text to the right.

Clear this checkbox if you want typing to overwrite existing text.

Use tab character (Options|Environment|Editor|Options)

Inserts a true tab character (ASCII 9) when you press Tab.

Clear this checkbox if you want to replace tabs with spaces.

The Tab Stops option determines the number of spaces used to replace a tab.

Optimal Fill (Options|Environment|Editor|Options)

Begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary.

Clearing checkbox will cause lines to contain more characters.

This option works only when the Use Tab Character option is on.

Backspace unindents (Options|Environment|Editor|Options)

If the cursor is on a on the first nonblank character of a line, when Backspace Unindents is

on the Backspace key aligns (outdents) the line to the previous indention level

off the Backspace key moves the cursor one character to the left and deletes it

If the cursor is on a tab and the Backspace Unindents is **off**, and Cursor through Tabs is

on the Backspace key moves the cursor one character to the left and deletes it, but treats null characters to the let of the tab as blank spaces

off the Backspace key moves the cursor one tab to the left and deletes it plus all leading null characters to the left of the tab

Cursor through tabs (Options|Environment|Editor|Options)

Causes the cursor to move uniformly through the line as you press arrow keys for horizontal movement.

Clear this checkbox if you want the cursor to jump several columns when moved over a tab.

- This setting has no effect unless Tab Stops option is checked.

Group undo (Options|Environment|Editor|Options)

See also Example

Causes the Edit|Undo command to reverse the effects of the last command plus all immediately preceding commands of the same type.

- Insertions, deletions, overstrikes, and cursor movements are each types of editing commands that can be grouped.
- The editor treats inserting a carriage return (pressing Enter) as an insertion followed by a cursor movement.

For example, the grouping of inserted characters stops when you press Enter because the type of editing has changed (you inserted characters, then moved the cursor).

- Clear this checkbox if you want the Edit|Undo command to reverse only the effect of a single editor command or keystroke.

Group Undo Examples

If Group Undo is on and you type OOPS then choose Undo, the entire word is deleted. If you select Edit| Redo after undoing OOPS, the entire OOPS group appears in the window all at once.

If Group Undo is off and you type OOPS, only the last typed character is undone when you choose Undo. You would need to choose Undo four times to undo the word OOPS. Only one letter will appear each time you choose Redo.

Cursor beyond EOL (Options|Environment|Editor|Options)

Lets you move the cursor beyond the EOL (end of line). If this checkbox is cleared, the cursor stops at the EOL and cannot be moved beyond it.

Cursor beyond EOF (Options|Environment|Editor|Options)

Lets you move the cursor beyond the EOF (end of file). If this checkbox is cleared, the cursor stops at the EOF and cannot be moved beyond it.

Undo after save (Options|Environment|Editor|Options)

Lets you perform an Edit|Undo command after the file has been saved.

If this checkbox is cleared, an Edit|Undo command cannot be performed after the file has been saved. The undo buffer is deleted each time the file is saved.

Keep trailing blanks (Options|Environment|Editor|Options)

Allows trailing blanks in lines to be stored as a part of the file. This means you should be able to cursor through them, and END should take you beyond them, not beyond the last word on the line. If this checkbox is cleared, trailing blanks are ignored and not saved as part of the file.

BRIEF regular expressions (Options|Environment|Editor|Options)

See also

Supports BRIEF-style regular expressions in editing operations such as search-and-replace. If this checkbox is cleared, GREP-like regular expressions are used, unless turned off in the Find Text dialog box.

Overwrite blocks (Options|Environment|Editor|Options)

Click this checkbox and the Persistent Blocks if you want selected text deleted as you type. Then, if you mark a block of text and type a letter, the letter you type replaces the entire marked block.

If you press...	Overwrite blocks will...
DEL or Backspace.	Clear the entire block of selected text
Any key or choose Edit|Paste.	Replace the entire block of selected text

- When this checkbox is cleared and Persistent Blocks is checked, text entered in a marked block is added at the insertion point.

Persistent blocks (Options|Environment|Editor|Options)

Allows marked blocks to remain selected until they are deleted or unmarked, or another block is selected.

- When this checkbox is cleared, if you move the cursor after a block is selected, the text does not stay selected.

Double Click line (Options|Environment|Editor|Options)

Lets you select a line by double-clicking the mouse. (The entire line is selected even if the line is blank.)

Clear this checkbox if you want double-clicking to select the word to the left of the insertion point.

- If you clear this checkbox and double-click when the cursor is:
- past the beginning of the line, the first word on the line is selected.
- past the end of the current line, the last word on the line is selected.
- on a blank line, nothing happens.

Block Indent (Options|Environment|Editor|Options)

Lets you specify the number of characters blocks are indented when the block indent and block outdent commands are used. The number can be from 1 to 16. See Keyboard for more information.

Undo Limit (Options|Environment|Editor|Options)

See also

Lets you specify the number of keystrokes that you can undo. The number can be from 0 to 32767. The default is 32767.

Tab Stops (Options|Environment|Editor|Options)

See also

Lets you specify multiple tab stops. Valid entries correspond to a list of numbers in which each number must be greater than the number to its left. Tab Stops are set on each value. When the list of tab stops has been exhausted, tabs are set to the:

```
<last number entered> + (<last number entered> - <second to last number entered>)
```

Example

The values `3 6 9 12 25` would set tab stops at columns 3, 6, 9, 12, and 25. After column 25, each tab pressed will move the cursor 13 characters to the right.

Editor | File (Options|Environment|Editor|File)

The Editor|File options let you tailor the behavior of the IDE editor.

The options are:

Create Backup Files

Preserve Line Ends

Default Extension

Original Path

Mirror Path

Backup Path

Keystroke mapping

Reserved Words

Create Backup Files (Options|Environment|Editor|File)

Automatically creates a backup of the source file loaded in the active Edit window when you choose File|Save. The backup file has the extension .BAK.

The Mirror, Original, and Backup paths are all independent. The only relationship between each of these paths is the Create Backup files checkbox.

When this checkbox is cleared, these options are ignored. The next time you check Create Backup, however, these options are restored to their current values.

Preserve Line Ends (Options|Environment|Editor|File)

Saves files with their original line ends. When this checkbox is cleared, files are saved with the Borland C++ default value for line ends.

Use this option to specify how the line ends are written when a file is saved: you can use the Borland C++ default value, or you can write the original line end of the file.

Line ends usually consist one of the following combination of characters:

LF

CR

LF CR

CR LF (Borland C++ default)

where LF = Line Feed (ASCII value 10) and CR = Carriage Return (ASCII value 13).

Default Extension (Options|Environment|Editor|File)

Tells the Borland C++ IDE which extension to use when you open or save a source file that has no extension.

Ending a file name with a period indicates that the file should have a blank extension; the Borland C++ IDE will not add one.

- Changing the extension here does not affect the history lists in the current desktop.

Original Path (Options|Environment|Editor|File)

Specifies a directory to store original versions (snapshots) of changed files. The original version is a copy of the file that was initially opened during your current session using the Borland C++ IDE.

This file is created when the first modification is made to the original file, then it is never updated. The full path of the file is saved, and directories will be created if necessary.

- This option only takes affect when the Create Backup Files option is checked.
- The Mirror, Original, and Backup paths are all independent. The only relationship between each of these paths is the Create Backup Files checkbox.
- When the Create Backup Files option is off, these options are unavailable. If the Create Backup Files option is turned back on, these options are restored to their current values.

Mirror Path (Options|Environment|Editor|File)

Specifies a directory to store the latest versions of a changed file. The full path of the file is saved, and directories will be created if necessary.

- This option only takes affect when the Create Backup Files option is checked.
- The Mirror, Original, and Backup paths are all independent. The only relationship between each of these paths is the Create Backup Files checkbox.
- When the Create Backup Files option is off, these options are unavailable. If the Create Backup Files option is turned back on, these options are restored to their current values.

Backup Path (Options|Environment|Editor|File)

Specifies a directory where backup files should be stored.

- Backup files are only saved when the Create Backup Files option is checked. If the Backup Path edit field is blank, the backup files are stored in the current directory with a .BAK extension. Otherwise, they are saved with the complete file name.
- The Mirror, Original, and Backup paths are all independent. The only relationship between each of them and the Create Backup Files checkbox.
- When the Create Backup Files option is off, these options are unavailable. If the Create Backup Files option is turned back on, these options are restored to their current values.

Keystroke Mapping (Options|Environment|Editor|File)

Lets you specify a KBD file to use to control keyboard behavior.

Reserved Words (Options|Environment|Editor|File)

See also

Lets you select which set of keywords are supported by Syntax Highlighting options.

Editor | Display (Options|Environment|Editor|Display)

The Editor|Display options let you tailor the behavior of the IDE editor.

The options are:

Horizontal Scroll Bar
Vertical Scroll Bar
BRIEF Cursor Shapes

Visible gutter
Gutter width
Visible Right Margin
Right Margin

Font
Size

Sample Text

Horizontal Scroll Bar **(Options|Environment|Editor|Display)**

Displays horizontal scroll bars in active Edit windows.

Clear this checkbox if you want to hide horizontal scroll bars in active Edit windows.

Vertical Scroll Bar (Options|Environment|Editor|Display)

Displays vertical scroll bars in active Edit windows.

Clear this checkbox if you want to hide vertical scroll bars in active Edit windows.

BRIEF Cursor Shapes (Options|Environment|Editor|Display)

Toggles the cursor shapes in the Edit window.

The default Windows cursor is a vertical bar between characters. Turning the BRIEF Cursor Shapes option on causes the flashing text cursor to become an underline. This underline cursor will enlarge when it is located on virtual white space in the Edit window.

Visible gutter (Options|Environment|Editor|Display)

See also

Displays a margin on the left edge of the Edit window that displays symbols which indicate the state of each line in your source as you debug your program. Use Gutter width to customize its size.

- Any change you make takes effect immediately. Clicking or clearing this checkbox toggles the gutter display on or off in all open Edit windows, as well those opened subsequently.

Gutter width (Options|Environment|Editor|Display)

Select a positive decimal measurement (for example 16) to specify the gutter width. The default setting is 32.

- Changes you make here do not take effect until you open a new Edit window. The gutter width of any open Edit window retains the setting in effect when they were opened.
- Effects of this setting are visible only when Visible gutter is checked.

Visible Right Margin (Options|Environment|Editor|Display)

See also

Displays a light gray vertical line at the Right margin setting (default of 80). The margin will appear to the right of any character entered at the margin setting.

Clear this checkbox to hide the light gray vertical line.

- This is a visual setting only and does not prevent typing beyond the margin setting.

Right Margin (Options|Environment|Editor|Display)

See also

Lets you specify the Right Margin for Edit windows. Valid entries here are from 1 to 1024.

Font (Options|Environment|Editor|Display)

Lets you select the font used in the IDE editor. The editor supports only monospaced fonts. Only monospaced fonts currently installed on your system will appear in the Font list box.

Size (Options|Environment|Editor|Display)

Lets you select the point size for the font used in the IDE editor.

Sample text

The Sample display box shows you the current syntax highlighting settings for your Borland C++.

Default editor options

Option	Default	Classic	Brief	Epsilon
Auto Indent Mode	ON	ON	ON	ON
Backspace Unindents	ON	ON	ON	ON
Block Indent	3	3	3	3
BRIEF Cursor Shapes	OFF	OFF	ON	OFF
BRIEF Regular Expressions	OFF	OFF	ON	OFF
Cursor Beyond EOF	OFF	OFF	ON	OFF
Cursor Through Tabs	ON	ON	ON	ON
Double Click Line	OFF	OFF	OFF	OFF
Group Undo	OFF	OFF	ON	OFF
Insert Mode	ON	ON	ON	ON
Keep Trailing Blanks	OFF	OFF	ON	ON
Optimal Fill	ON	ON	ON	ON
Overwrite Blocks	ON	OFF	OFF	OFF
Persistent Blocks	OFF	ON	OFF	OFF
Tab Stops	3	3	3	3
Undo After Save	OFF	OFF	OFF	OFF
Undo Limit	32767	32767	32767	32767
Use Tab Character	ON	ON	ON	ON

Syntax Highlighting

See Also

The Options|Environment|Syntax Highlighting options let you customize the colors and styles used to denote syntax elements in your code. These options also tell the Borland C++ IDE editor how to display these elements.

The subtopic is:

Customize

The options are:

Use Syntax Highlighting

Syntax Extensions

Color SpeedSettings

Color SpeedSettings (Options|Environment|Syntax Highlighting)

See Also

The Color SpeedSettings allow you to use preset syntax highlighting color schemes. You can see the effect of each SpeedSetting by watching the sample window on the right.

Defaults

Press the Defaults button to use the default syntax highlighting color scheme.

Classic

Press the Classic button to use the Borland DOS IDE syntax highlighting color scheme.

Twilight

Press the Twilight button to use the Twilight syntax highlighting color scheme.

Ocean

Press the Ocean button to use the Ocean syntax highlighting color scheme.

Use Syntax Highlighting (Options|Environment|Syntax Highlighting)

See Also

When the Use Syntax Highlighting option is on, the editor displays your program code in the specified colors and attributes.

When this option is off, all of your program code appears in the color defined for the Plain Text element in the Element list box.

Syntax Extensions (Options|Environment|Syntax Highlighting)

See Also

By default, only files with a .C, .CPP, or .H extension display syntax highlighting. You might want to highlight other file types.

To change which files will be displayed with syntax highlighting, enter the desired file extensions in the Syntax Highlighting|Syntax Extensions input box. Any file extension, including one with wildcards (*), is valid. To enter multiple file extensions, place a semicolon (;) between each one.

Syntax Highlighting | Customize (Options|Environment)

See Also

Use the Syntax Highlighting|Customize options to set colors for specific code elements. The options you can set are:

Element

Color

Attribute

Default FG (foreground)

Default BG (background)

Sample

Element list box

See Also

You use the Syntax Highlighting|Customize Element list box to choose a syntax element whose colors you want to change. You can also click on an element in Sample display box to select an element.

The syntax elements whose colors and display attributes you can alter include:

Whitespace	Search Match
Comment	Execution Point
Reserved Word	Enabled Break
Identifier	Disabled Break
Symbol	Invalid Break
Scope Delimiter	Error Line
String	Plain Text
Integer	Client 1
Float	Client 2
Octal	Client 3
Hex	Client 4
Character	Client 5
Preprocessor	Client 6
Illegal Character	Client 7
Marked Block	Client 8

As you change the colors or underline attributes of a syntax element, the changes appear in the Sample code box.

Color

See Also

The Syntax Highlighting|Customize|Color matrix displays the possible color choices for elements of code.

You can set the foreground or background colors for any element. As soon as you select a color for an element, the change appears in the Sample display box.

FG	The rectangle containing the letters FG displays the current element's foreground color.
BG	The rectangle containing BG displays the element's background color.
FB	A rectangle containing the letters FB displays the current element's foreground and background color (where the same color is assigned to both).

This option works differently than the Default FG and Default BG options that set the colors of an element to the defaults defined in your Windows Control Panel.

Attribute

See Also

Use the Syntax Highlighting|Customize|Attribute options to choose the attributes of a code element.

Select the code element you wish to change by clicking on the element in the Element list box. Then, click the attribute you want to use. As soon as you choose the attribute, it is reflected in the Sample display box.

Bold

Use the Bold option to make the specified code element display in boldface.

Italic

Use the Italic option to make the specified code element display in italics.

Underline

Use the Underline option to make the specified code element display underlined.

Default FG (foreground color)

See Also

You use the Syntax Highlighting|Customize|Default FG option to set the foreground color of the element currently selected in the Element list box to your Windows system text color.

To change the Windows system colors, use the Control Panel in the Program Manager.

Default BG (background color)

See Also

You use the Syntax Highlighting|Customize|Default BG option to set the background color of the element currently selected in the Element list box to your Windows system background color.

To change the Windows system colors, use the Control Panel in the Program Manager.

Sample display box

See Also

The Sample display box shows you the current syntax highlighting settings for your Borland C++ Edit window.

When you change these settings using Syntax Highlighting|Customize options, the colors in the Sample display box change to reflect any changes you have made in the options.

Use the scroll bars to see other sections of the sample code to see the effect on types of elements not currently visible in the Sample display box.

You can select which code element you want to customize the colors for by clicking on it on the Sample display box, or by selecting it in the Element list box.

Using syntax highlighting

You use syntax highlighting to control how code is displayed. Setting the different syntax elements to different colors makes it easier for you to quickly identify parts of your code.

Some of the tasks you can perform with syntax highlighting include:

Activating syntax highlighting

Selecting a syntax element

Changing the syntax highlighting colors

Setting the background color

Setting the foreground color

Setting text attributes

Setting up client areas

Most of these tasks are performed using the Syntax Highlighting options section of the Environment options settings dialog.

Activating syntax highlighting

See Also

To turn syntax highlighting on, you choose Use Syntax Highlighting option on the Syntax Highlighting options section of the Environment options settings dialog.

Selecting a syntax element

See Also

To select a Syntax Element to customize the colors for, either:

- Choose an element from the Element list box.
- Click on an element In the Sample Code display box.

Changing the syntax highlighting colors

See Also

To alter syntax highlighting colors for parts of your code, change the settings in the Syntax Highlighting options.

As you change the colors or attributes, the changes appear in the Sample Code display box.

Setting the background color

See Also

With the mouse

Right-click on the background color you want in the Color matrix.

The letters BG in the color rectangle indicate that the background is now set to that color.

From the keyboard

1. Press Tab (or Alt+C) to go to the Color matrix.
2. Press the arrow keys or Spacebar to select the background color you want.
3. Press B.

The letters BG in the color rectangle indicate that the background is now set to that color.

To use the Windows system background color for the current syntax element's background, choose the Default BG option.

Setting the foreground color

See Also

With the mouse:

Left-click the foreground color you want in the Color matrix.

The letters FG in the color rectangle indicate that the foreground is now set to that color.

From the keyboard:

1. In the Color matrix, press the arrow keys or Spacebar to select the foreground color you want.
2. Press F. The letters FG in the color rectangle indicate that the foreground is now set to that color.

To use the Windows system text color for the current syntax element's foreground, choose the Default FG option.

Setting text attributes

See Also

Set the normal, bold, italic, or underline attribute: choose the Bold, Italic, or Underline options.

Setting up client areas

See Also

Among the elements for which you can specify syntax highlighting are eight user-defined lists in which you can specify your own keywords, functions, or other language elements that you want highlighted. These elements are stored in token (.TOK) files.

Two token files are shipped with Borland C++: C.TOK and C32.TOK. You select the one you want to use in Reserved Words option in the Editor|File settings.

You can modify these files to include your own keywords. To add keywords to a token file:

1. Open the token file you want to modify.
2. At the end of the file add the heading `[Client 1]`.
3. Start a new line and enter the new keyword.
4. Enter each additional keyword, making certain that there is a carriage return after each keyword you add.

If you want to create a new user-defined area, place a new heading (as in step 2), incrementing the number by 1. You can have up to 8 of these areas.

- You must exit the IDE and restart Borland C++ for your changes to take effect.

Example

```
[Client 1]
LocateWinFunction
LocateDOSFunction

[Client 2]
Icon
Bitmap
ListBox
```

SpeedBar options

Add-on Products

The Options|Environment|SpeedBar options control what commands can be accessed from the IDE SpeedBar.

The subtopics are:

Customize

The options are:

SpeedBar Options

Orientation

SpeedBar | SpeedBar Options (Options|Environment|SpeedBar)

The SpeedBar|SpeedBar Options control the operation of the SpeedBar.

Hide SpeedBar
Check Hide SpeedBar to prevent the SpeedBar from being displayed in the Borland C++ IDE desktop.

Use Flyby Help Hints
Check Use Flyby Help Hints to display status line help hints when you move the mouse pointer over a SpeedBar button.

When this option is off, you must press the left mouse button on the SpeedBar button to view a help hint. To get a help hint without having the SpeedBar action take place, press the left mouse button and move the pointer away from the SpeedBar button before releasing it.

Big Buttons
Check Big Buttons if you want to use large buttons on the SpeedBar.

SpeedBar | Orientation (Options|Environment|SpeedBar)

The SpeedBar|Orientation options control the placement of the SpeedBar on the Borland C++ IDE desktop.

Choose...	To display the SpeedBar...
Top	at the top of the Borland C++ IDE desktop.
Bottom	at the bottom of the Borland C++ IDE desktop.
Left	on the left side of the Borland C++ IDE desktop.
Right	on the right side of the Borland C++ IDE desktop.
Floating	in a floating window that you can size and place anywhere on top of your Windows desktop.

- You can also drag the Speedbar with the mouse to change its size and position.

SpeedBar | Customize (Options|Environment|SpeedBar|Customize)

The SpeedBar|Customize options control which commands can be accessed from the IDE SpeedBar.

Window

Use the Window list to select the IDE desktop window you want to customize the SpeedBar for. The windows for which you can customize the SpeedBar include the Edit window, Project Hierarchy, and Message window.

Available buttons

The Available Buttons list displays the commands (and the icons associated with them) that you can use on your SpeedBar. This list excludes any buttons that are currently in the Active Buttons list.

Active buttons

The Active Buttons list displays the commands (and the icons associated with them) that are currently on the SpeedBar for the IDE window currently selected in the Window list box. This list excludes any buttons that are currently in the Available Buttons list. The button that appears at the top of the list appears on the far left of the SpeedBar; the last button appears on the far right.

Customizing a SpeedBar

- Use the Copy Layout button to copy the SpeedBar options from the window currently selected in the Window list box to another window.
- Use the Restore Layout button to restore the default SpeedBar options of the one currently selected in the Window list box.
- Use the Separator button to place a blank space above the selected button in the Active Buttons list for the SpeedBar you are customizing.
- Use the Left Arrow button to move the button currently selected in the Active Buttons list and place it in the Available Buttons list.
- Use the Right Arrow button to move the button currently selected in the Available Buttons list and place it in the Active Buttons list.
- Use the Up Arrow button to move the selected button in the Active Buttons list up one position in the SpeedBar.
- Use the Down Arrow button to move the selected button in the Active Buttons list down one position in the SpeedBar.

Copy Speedbar Layout dialog box (Options|Environment|SpeedBar|Customize)

Use the Copy Speedbar Layout dialog box to specify which windows the current SpeedBar layout should be copied to. The name of the window currently selected in the Window list of the Speedbar|Customize settings will be grayed out.

Scripting (Options|Environment)

The Options|Environment|Scripting options let you specify options to control scripting behavior.

The options are:

Stop at Breakpoint
Diagnostic Messages
Startup Scripts
Script Path

Stop at Breakpoint (Options|Environment|Scripting)

Tells the IDE to stop script execution if the script file contains the script breakpoint statement. When this option is checked, script execution stops at a line if it contains the **breakpoint** statement and the IDE passes control to the script debugger which opens the Script Breakpoint tool.

Clear this checkbox if you want script processing to complete without interruption regardless of whethera script file contains one or more **breakpoint** statements.

Default is on.

Diagnostic Messages (Options|Environment|Scripting)

Displays all script processor messages in the Script tab of the IDE Message window when you run a script. These messages will appear in addition to the script messages generated by the script print command.

Default is off.

Startup Scripts (Options|Environment|Scripting)

See also

Specifies one or more script (.SPP or .SPX) files that run automatically when you start the Borland C++ IDE. The IDE executes these scripts immediately after running the IDE startup script (STARTUP.SPP).

Use a space to separate each script name. To specify script parameters, enclose the script name and its arguments in quotation marks ("). The following example specifies three scripts: two without arguments and a third with an argument.

```
MyStartup DisplayCurProj "Ascript Param1"
```

- Use Script Path to tell the IDE in which directories to look for script files.

Script Path (Options|Environment|Scripting)

Tells the Borland C++ IDE one or more directory locations in which to look for script (.SPP or .SPX) files. When loading a script, the IDE searches every entry on the path for a file with the .SPX extension. If that fails, the same directories will be searched a second time for files with the .SPP extension.

You can specify a fully qualified path, a relative path, or multiple locations. To specify multiple locations, separate each path with a semicolon (;).

- Start the path with `.;` to cause the directory to be searched first.
- Be sure not to insert any spaces before a path. Doing so will stop the search at the previous path.

Examples:

```
c:\bc5\script
..\..\script
.;c:\myscripts\;c:\bc5\script;..\..\script
```

Process Control (Options|Environment)

The Options|Environment|Process Control options let you determine how processes are executed.

The options are:

Build Process

Status Box

Build priority boost

Update interval

IDE priority

Build Process (Options|Environment|Process Control)

Click...	If you want...
Asynchronous	to compile and link using background processing and make system resources available for other purposes during build time. For example, click this checkbox if you want to run other processes (such the Browser, text search in the Edit Window, or Grep) while you build your project. Otherwise, clear this checkbox if you want to commit all system resources to the build process.

- You cannot use the debugger to step through code during build time.

Click...	If you want...
Beep on Completion	your computer to beep when your program compiles successfully.

Status Box (Options|Environment|Process Control)

Click...	If you want the status box to display...
None	no messages
Status Only	only information such as compiling or success on the status line
Full with statistics	all messages (warnings and errors) generated during the build process including a count of lines processed

Build priority boost (Options|Environment|Process Control)

Sets the level of system resources that will be shared with other processes during build time.
Choose from -2 to 2, where:

-2 shares the maximum amount of available resources.

2 commits the maximum amount of resources to the build process.

You can also set the priority boost with the slide-bar on the status box that displays during build time.
The level supplied here will be used as the slide-bar's initial setting.

- This option has no effect if you clear the Asynchronous checkbox.

Update interval (Options|Environment|Process Control)

Determines how often to refresh the messages shown in the status box that displays during build time. A small interval (more frequent) may slightly increase build time.

IDE priority (Options|Environment|Process Control)

Sets the amount of system resources that will be shared with processes running outside of the Borland C++ IDE, such as tools, DOS shells, and other Windows applications.

Click...	If you want to allocate system resources...
Low	in favor of processes outside the IDE.
Normal	equally between the IDE and other processes.
High	in favor of the IDE.
Realtime	exclusively to the IDE.

For example:

- Click **Low** to increase the performance of an application running outside the IDE, such as Paradox for Windows, while running a process in the IDE such as text search or debugging.
- Click **Normal** if you want the operating system to regulate performance between all active processes.
- Click **High** to increase IDE performance while running a process outside the IDE.
- Click **Realtime** to devote all system resources to an active process in the IDE and temporarily halt all other processes.

Tip: Be sure to set the IDE Priority to **Normal** when you use the integrated debugger. Since the debugger runs on a different thread than does the IDE, setting the IDE Priority to Normal ensures that both processes get equal status.

Preferences Options (Options|Environment|Preferences)

Environment Options

Let you set source tracking options and control what desktop information is saved when you exit the Borland C++ IDE, build or make a project, use a transfer tool, run the integrated debugger, close a project, save a project, or switch to another project.

The options are:

Save In Desktop	Auto-Save
Message Window	Source Tracking
Enable Filename Completion	

Save In Desktop (Options|Environment|Preferences)

See also Preferences

Let you specify what parts of the desktop you want saved when you exit the Borland C++ IDE, build or make a project, use a transfer tool, run the integrated debugger, close a project, save a project, or switch to another project.

The options are:

History Lists

When the History Lists option is on, history lists are saved across sessions. Default = ON.

Watch Expressions

When the Watch Expressions option is on, watch expressions are saved across sessions. Default = ON.

Breakpoints

When the Breakpoints option is on, breakpoints are saved across sessions. Default = ON.

Open Windows

When the Open Windows option is on, the position and content of any open windows are saved across sessions. Default = ON.

Closed Windows

When the Closed Windows option is on, the list of closed windows that appears at the bottom of the File menu is saved across sessions. Default = ON.

Auto Save **(Options|Environment|Preferences)**

Preferences

Let you specify what is automatically saved when you exit the Borland C++ IDE, use a transfer tool, run the integrated debugger, close a project, save a project, or switch to another project.

The options are:

Editor Files

Environment

Desktop

Project

Messages

Editor Files (Options|Environment|Preferences)

Preferences

When the Editor Files option is on, the Borland C++ IDE automatically saves the modified file in the active Edit window when you exit the Borland C++ IDE, build or make a project, use a transfer tool, run the integrated debugger, close a project, save a project, or switch to another project.

Default = OFF

Environment (Options|Environment|Preferences)

Preferences

When the Environment option is on, the Borland C++ IDE also saves the settings to disk when you exit the Borland C++ IDE, build or make a project, use a transfer tool, run the integrated debugger, close a project, save a project, or switch to another project.

If you turn the Environment option off, you must use the Options|Save command to save the environment settings to disk.

Default = ON

Desktop (Options|Environment|Preferences)

Preferences

When the Desktop option is on, the IDE saves your desktop files in the file PRJNAME.DSW.

Each project file has a corresponding desktop file that contains state information about the associated project.

Desktop information saved includes the following:

- context information for each file in the project (position in the file, location of the window on the screen, etc.)
- history lists for various input boxes
- layout of the windows on the desktop

If you turn the Desktop option off, you must use the Options|Save command to save the desktop settings to disk.

Default = ON

Project (Options|Environment|Preferences)

Preferences

When the Project option is on, the Borland C++ IDE saves your project, autodependency, and default style sheet settings on exit and restores them when you return to the IDE and load that project.

The project file contains the information required to build the executable targets of the project (FILENAME.EXE or FILENAME.DLL).

The build information consists of the following:

- source files and libraries
- compiler options
- include, library, and output paths
- linker options
- build options and attributes
- transfer tools

Default = ON

Messages (Options|Environment|Preferences)

Preferences

Click this option if you want the Borland C++ IDE to automatically save to disk the contents of the Message window when you close the current project or exit the IDE.

Default = ON

Message Window (Options|Environment|Preferences)

Preferences

Lets you specify options about the display of the Message window.

The options are:

Save Duplicate Messages

Clear Before Make/Build

Save Duplicate Messages (Options|Environment|Preferences)

Preferences

Click Save Duplicate Messages if you want the Borland C++ IDE to save error messages already contained in the Message window.

- Only transfer items that prompt for a command line will have their messages saved. They will be saved only when the command line for the tool being run has been modified, (i.e. when you are prompted from transfer items for a command line, like GREP). In every other case, the messages will be removed. Simple compiles do not save messages.
- To change the command line for internal tools, use the Options|Tools command.

Default = OFF

Clear Before Make/Build (Options|Environment|Preferences)

Preferences

Click Clear Before Make/Build to clear the Buildtime tab of the Message window before a make or build. After the make or build, the Buildtime tab will only contain information about the most recent make or build.

If this option is off, the Buildtime tab will contain information from all previous makes or builds, including the most recent one.

Source Tracking (Options|Environment|Preferences)

Preferences

Indicates if the Borland C++ IDE opens a new Edit window or uses the current one as you step through your code to a file that is not already loaded in an Edit window.

Option	Description
New Window	Opens a new Edit window whenever the IDE encounters a source file that is not already loaded.
Current Window	Replaces the contents of the active Edit window whenever the IDE loads a new file. (Default)

The IDE uses the specified Source Tracking setting whenever you open an Edit window. For example, an Edit window opens when you

- double-click a source node in the Project Tree window.
- double-click a message in the Message window.
- choose Edit Source from the Browser window SpeedMenu.
- choose the Search|Locate symbol command.
- choose Edit|Buffer List.

Enable Filename Completion (Options|Environment|Preferences)

Preferences

Allows directory and file name completion in the File Name input box of the Open a File dialog.

When Enable Filename Completion is on, you can type in a few characters of the directory and file name and press the space bar to force Borland C++ to complete the name for you.

- If there is no match, the system will beep and place a space in the File Name input box.
- If there is one match, the directory and file name will be inserted in the File Name input box.
- If there is more than one match, a list will be displayed. You can choose the directory and file you want.

Fonts (Options|Environment)

The Options|Environment|Fonts options let you customize the character font of displayed text for each view in the Borland C++ IDE:

View

Text Type

Font

Size

Attributes

Color

To customize how text is displayed:

1. Select a view and the type of text.
2. Choose a font, size, attribute, and color.

View (Options|Environment|Fonts)

Select the view that you want to customize. You can specify individual settings for the Message, Project, Watches, Breakpoints, CPU, Call Stack, and Inspector views.

- To customize text displayed in the editor, choose Options|Environment|Editor Display.

Text Type (Options|Environment|Fonts)

Select the type of text you want to display in the selected view. The number of available text types differs for each view. For example, only one type of text is available for Breakpoints and Watches, whereas several types are available for the CPU view, such as Label, Source, Assembly Code, and so forth.

Font (Options|Environment|Fonts)

Sets the character font for the selected view and text type.

Size (Options|Environment|Fonts)

Sets the character size for the selected view and text type.

Attribute (Options|Environment|Fonts)

Bold
Makes the specified text display in boldface.

Italic
Makes the specified text display in italics.

Underline
Makes the specified text display underlined.

Color (Options|Environment|Fonts)

Click the box that displays the color you want to use for the selected view and text type.

Debugger Options (Options|Environment|Debugger)

See also

The Options|Environment|Debugger options let you configure the integrated debugger and customize the way you debug your program.

The options on this page are:

Run Arguments

Source Path

In addition to these options, the Debugger Behavior page contains additional debugger-related options.

Run Arguments (Options|Environment|Debugger)

See also

Enter any command-line options you want to pass to the application being debugged.

In addition to this option, you can also specify program arguments if you use the Debug|Load command to begin a debugging session.

Source Path (Options|Environment|Debugger)

See also

Enter a directory location of any additional source files that you want the application you are debugging to include. You can specify a fully qualified path, a relative path, or multiple source paths. To specify multiple source paths, separate each path with a semicolon (;).

Examples:

```
c:\bc5\examples\myprogram
..\..\include
c:\bc5\examples\myprogram;..\..\include
```

- Be sure not to insert any spaces before your source path. Doing so will stop the search at the previous path.

Debugger | Debugger Behavior (Options|Environment|Debugger|Debugger Behavior)

See also

The Debugger|Debugger Behavior options let you configure the integrated debugger.

The options are:

Make errors prevent debugging

Show child window whenever child runs

Kill child processes when project is closed

Allow multiple CPU views

Debug spawned processes

Leave tracks in source where child stopped

Record Windows messages sent

Record Windows messages posted

Do not save files or prompt when debugging

Do not promt when attempting to run 16-bit applications

Run to ... on startup

Errors and messages

Make errors prevent debugging (Options|Environment|Debugger|Debugger Behavior)

See also

Prevents the debugger from running if the program fails to build.

Clear this checkbox if you want to continue debugging the last successful build even after your most recent changes caused a build failure.

Show child window whenever child runs (Options|Environment|Debugger|Debugger Behavior)

See also

Brings the application's main window (or console window) to the foreground whenever the process runs. This is particularly useful if the application you are debugging requires user input.

Kill child processes when project is closed (Options|Environment|Debugger|Debugger Behavior)

See also

Terminates all processes running in the IDE when you close the project. Clear this checkbox if you want continue debugging after you close the project.

Allow multiple CPU views (Options|Environment|Debugger|Debugger Behavior)

See also

Lets you view multiple CPU windows at the same time. Each time your application spawns a new process, a new CPU Window opens.

Clear this checkbox if you want only one CPU window open at a time. Each process viewed replaces the contents of the current CPU window.

Debug spawned processes (Options|Environment|Debugger|Debugger Behavior)

See also

Lets you debug a process spawned by the current process you are debugging.

Check this box if you want the debugger to pause when a child process is spawned. The debugger pauses at the entry point inside the child process.

Clear this checkbox if you do not want the debugger to stop when a child process is spawned. You may find it useful to keep this checkbox cleared if you are debugging an application that

- calls the Windows Help engine.
- issues a Send command to the current mail server.
- is a container of an embedded OLE object that spawns a process requesting the OLE server for processing.

Leave tracks in source where child stopped (Options|Environment|Debugger|Debugger Behavior)

See also

Displays markers in the gutter on the left of the Edit Window. The markers indicate the source line where a process stopped running.

Record windows messages sent (Options|Environment|Debugger|Debugger Behavior)

See also

Records the window messages that are sent to the program you are debugging. The messages appear in the Runtime tab of the Message window.

Record windows messages posted (Options|Environment|Debugger|Debugger Behavior)

See also

Records the window messages that are posted to the program you are debugging. The messages appear in the Runtime tab of the Message window.

Do not save files or prompt when debugging (Options|Environment|Debugger|Debugger Behavior)

Clear this checkbox if you want to be prompted to save unsaved changes to your program files each time you start a debugging session in the IDE.

Do not prompt when attempting to run 16-bit applications (Options|Environment|Debugger|Debugger Behavior)

Clear this checkbox if you want the IDE to issue a message that you cannot use the integrated debugger to debug a 16-bit application.

Run to ... on startup (Options|Environment|Debugger|Debugger Behavior)

See also

Executes your program at full speed until it reaches the line in the source file that contains the function or symbol initialization that you specify in this field. This command works in conjunction with the Debug|Load command.

The function or symbol you specify must evaluate to an executable code address. For example, *OwlMain*, *WinMain*, or *main* are permissible, whereas values such as global variables and static functions are not.

If you leave this field blank, the Debug|Load. command starts the debugger at the location of the program's startup code. If the debugger cannot locate the item you specify, the program runs to termination.

- This option does not behave the same as the Run to menu command.

Errors and messages (Options|Environment|Debugger|Debugger Behavior)

See also

Let you choose how run-time messages display:

All notices only in pop-up windows. Displays each message in a dialog box and the process pauses until you press OK. This option may be useful when you are trying to find the cause of a particular message, but in most cases it proves to be time consuming as it requires you to dismiss a dialog box after the generation of each message.

All notices only in status bar. Displays all messages only in the status bar on the IDE desktop. This option requires the least amount of intervention, but messages too long to fit in the status bar are truncated.

Errors pop up, messages in status bar. Displays only error level messages in a dialog box, such as those that appear if you try to load a process that does not exist or inspect an undeclared variable. All other warnings appear only in the status bar of the IDE.

- All debugger messages also appear in the Runtime tab of the Message Window.

Project View Options (Options|Environment|Project View)

See also

Use the Options|Environment|Project View options to specify what types of nodes appear in the project tree and what information appears with each node.

The options are:

Build Translator

Choose the Build Translator option to display the translator associated with the node.

Code Size

Choose the Code Size option to display the total size in bytes of code segments. This information appears only after the node has been built.

Data Size

Choose the Data Size option to display the size (in bytes) of the data segment. This information appears only after the node has been built.

Description

Choose the Description option to display a description of the node. Type the description using the Edit node attributes dialog box from the Project Window SpeedMenu.

Location

Choose the Location option to display the path to the file associated with the node.

Name

Choose the Name option to display the name of the file associated with the node.

Number of Lines

Choose the Number of Lines option to display the number of lines in the file associated with the node. This information appears only after the node has been built.

Node Type

Choose the Node Type option to display the type associated with a node. Some node-types are assigned by the system such as [AppExpert] and[SourcePool] but most node-types are a reflection of a file extension: [.cpp], [.exe], [.def], etc.

Every node in a project is an instance of a node type. It's the node type that determines which viewers and translators will be available for instances of that node-type.

- You can manipulate which viewers and translators are available through Options|Tools in the Advanced dialog box under Applies to, Translate from, and to edit fields.

The Project Manager and MAKE engine look to the node-type field to determine how to deal with each node. For example, the node name might be 'myfile.cxx' but if the node-type is [.cpp] the file will be treated as if it had a .cpp extension and all the C++ translator and viewers will be available to the 'myfile.cxx' node.

- You can manipulate an individual node's name and type through the Project Window's SpeedMenu Edit node attributes command.

Style Sheet

Choose the Style Sheet option to display the name of the Style Sheet attached to a node.

Output

Choose the Output option to display the name of the file (and its path) that is created when the node is translated. For example, a .CPP node creates an .OBJ file.

Show Runtime Nodes

Choose the Show Runtime Nodes option to display the standard libraries and startup code used when a target is built.

Show Project Node

Choose the Show Project Node option to show the top-level node associated with the entire project. The project node is built when you choose Project|Build All.

- All targets are dependents of the project node.