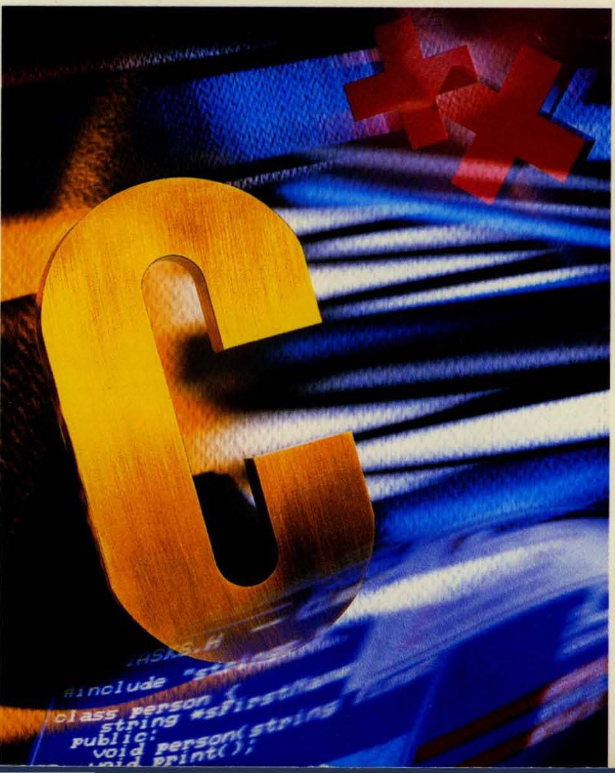
2.0

BORLAND® C++

USER'S GUIDE



BORLAND

Borland® C++ Version 2.0

User's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.
10 9 8 7 6 5 4

R1

C O N T E N T S

T A B L E S

F I G U R E S

INTRODUCTION

If you haven't already done so, read the introduction, Chapter 1 ("Installing Borland C++"), and Chapter 2 ("Navigating the Borland C++ manuals") in *Getting Started* for information on the overall organization of the Borland C++ manuals. Those chapters tell you about many of the highlights of Borland C++, how to install Borland C++, and how to use the manuals most effectively.

This book, the *User's Guide*, contains reference-style information on the Programmer's Platform (the IDE), using Borland C++ to write a Windows application, the Project Manager, the integrated editor, the command-line compiler, and some of the many utilities included with Borland C++. See the introduction and Chapter 2, "Navigating the Borland C++ manuals," in *Getting Started* for information on how to most effectively use the Borland C++ documentation set.

Here is a breakdown of the chapters in this book:

Chapter 1: The Programmer's Platform introduces the features of the Programmer's Platform, giving information and examples of how to use the IDE to full advantage. It includes information on how to start up and exit from the IDE.

Chapter 2: Menus and options reference provides a complete reference to the menus and options of the IDE.

Chapter 3: Building a Windows application tells you what you need and how to pull it together to write an application for Microsoft's Windows.

Chapter 4: Managing multi-file projects tells how to use the Project Manager to manage multi-file projects.

Chapter 5: The editor from A to Z provides a complete reference to the editor.

Chapter 6: The command-line compiler tells how to use the command-line compiler. It also explains configuration files.

Chapter 7: Utilities describes some of the utility programs that come with Borland C++.

Appendix A: Precompiled headers describes some of the utility programs that come with Borland C++.

Typefaces used in this book

The typefaces used in this manual are described in *Getting Started*.

C H A P T E R

1

The Programmer's Platform

Borland's Programmer's Platform, also known as the integrated development environment or IDE for short, has everything you need to write, edit, compile, link, and debug your programs. For example, it provides

- multiple, movable, resizable windows
- mouse support
- dialog boxes
- cut-and-paste and cut-and-paste commands
- full editor undo and redo
- examples ready to copy from Help
- a built-in assembler
- quick transfer to other programs (like TASM) and back again
- an editor macro language

The IDE runs in two modes: protected and real. Under "Starting up and exiting," you'll find a description of how to start up the IDE in either mode, and what the differences are. Since the IDE works the same in either mode, this chapter, and Chapter 2 (the menu reference) don't address the differences and the implications for your programs.

This chapter is divided into two main sections: "Starting up and exiting" tells you how to enter and exit the IDE; "The components," starting on page 8, discusses the generic components that comprise the IDE. Chapter 2, starting on page 25, provides a reference to each menu item and dialog box.

Starting up and exiting

You can run the IDE in either real or protected mode. You can use protected mode if you have a 286, 386, or i486 machine with 640K of conventional RAM and at least 576K of extended or (simulated or real) expanded memory. Otherwise, use real mode.

Note that, although you may be running Borland C++ in protected mode, you are still generating applications to run in real mode. The greatest advantages to using Borland C++ in protected mode are:

- both the compiler and your application have *much* more room to run than if you were running Borland C++ in real mode
- the linker runs faster

Running in real mode

To invoke the IDE in real mode, type `BC` at the DOS prompt; you can follow it with one or more options.

Running in protected mode

Running Borland C++ in protected mode requires a small amount of preparation. It involves interaction between three files: BCX.EXE, BCX.OVY, and TKERNEL.EXE. BCX.EXE loads TKERNEL and BCX.OVY, which is the protected-mode version of the IDE. Although BCX.EXE loads these files automatically, so that you don't need to be concerned with invoking them yourself, they do both need to be on the path or in the BCX.EXE startup directory so it can find them.

Once you've verified that the correct directories are on the path, running Borland C++ in protected mode is as simple as running it in real mode; the syntax is identical except for using BCX in the place of BC.

The options and menus are identical to those for BC; therefore, for the remainder of this chapter, when we mention the IDE we mean both BC and BCX (unless specifically called out otherwise).

BCX.EXE loads TKERNEL each time you invoke BCX. You can save some loading time by preloading TKERNEL; before running BCX, type

```
TKERNEL hi=yes
```

on the DOS command line. This has the added benefit of storing most of TKERNEL in extended memory, freeing more conventional memory for your application. When you are through with your Borland C++ session, type

```
TKERNEL rem
```

to remove TKERNEL.

Windows and protected mode

You can also run the protected mode versions of the command-line compiler and TLINK under Windows using the same procedure.

You can use the protected mode version of the IDE while running Windows. To do so, first load TKERNEL.EXE with the command:

```
TKERNEL hi=yes kilos=nnnn
```

where *nnnn* is the number of Kbytes to be managed by the kernel. We suggest kilos=2048. The remaining extended memory is available for Windows and other programs. Then run Windows in standard mode (type the command `WIN /s`). With Windows in standard mode, you can't run the IDE in a virtualized DOS window, but only as a full screen.

You can only run the protected mode IDE in Windows standard or real mode, not in enhanced mode.

Command-line options

The command-line options for Borland C++'s IDE are: **/b**, **/d**, **/e**, **/h**, **/l**, **/m**, **/p**, **/rx**, **/s**, and **/x**. These options use this syntax:

BC | BCX [*option* [*option...*]] [*sourcename* | *projectname* [*sourcename*]]

where *sourcename* is any ASCII file (default extension assumed), *projectname* is your project file (it *must* have the .PRJ extension), and *option* can be one or more of the options.

The /b option

The **/b** option causes Borland C++ to recompile and link all the files in your project, print the compiler messages to the standard output device, and then return to the operating system. This option allows you to invoke Borland C++ from a batch file so you can automate builds of projects. Before the build, Borland C++ will load a default project file or one given on the command line.

Borland C++ determines what .EXE to build based on the project file or the file currently loaded in the Editor if no project file is found.

Enter the `BC` or `BCX` command with either `/b` alone or the project file name followed by `/b`. For example,

```
BC /b
BC myproj.prj /b
```

Unless a project file is loaded, you can specify the name of a program to be compiled and linked on the command line. Type in the program name after the `BC` or `BCX` command, followed by `/b`:

```
BC myprog /b
```

The /d option

The **/d** option causes Borland C++ to work in dual monitor mode if it detects appropriate hardware (for example, a monochrome card and a color card); otherwise, the `/d` option is ignored. Use dual monitor mode when you run or debug a program, or shell to DOS (**File | DOS** Shell).

If your system has two monitors, DOS treats one monitor as the active monitor. Use the DOS MODE command to switch between the two monitors (`MODE CO80`, for example, or `MODE MONO`). In dual monitor mode, the normal Borland C++ screen will appear on the inactive monitor, and program output will go to the active monitor. So when you type `BC /d` or `BCX /d` at the DOS prompt on one monitor, Borland C++ will come up on the other monitor. When you want to test your program on a particular monitor, exit Borland C++, switch the active monitor to the one you want to test with, and then issue the `BC /d` or `BCX /d` command again. Program output will then go to the monitor where you typed the `BC` or `BCX` command.

Keep the following in mind when using the **/d** option:

- Don't change the active monitor (by using the DOS MODE command, for example) while you are in a DOS shell (**File | DOS** Shell).
- User programs that directly access ports on the inactive monitor's video card are not supported, and can cause unpredictable results.
- When you run or debug programs that explicitly make use of dual monitors, do not use the Borland C++ dual monitor option (`/d`).

The /e and /x options

*Of the three options, **/e**, **/x**, and **/rx**, **/x** is the best to use, followed by **/e**, with **/rx** the third best.*

Normally, Turbo C++ swaps to a hard disk when allocating memory. The **/e** option tells Borland C++ to swap to expanded memory, the **/x** option tells Borland C++ to swap to extended memory. The **/e** option is enabled by default. The syntax for these two options is as follows:

/e[=*n*]

where *n* equals the number of pages of expanded memory that you want the IDE to use for swapping. A page is 16K.

/x[=[*r*][,*n*]]

where *n* is the number of kilobytes of extended memory that you want the IDE to use for swapping and *r* is the number of kilobytes of extended memory to reserve for other programs.

➡ You can also use these options with the protected-mode version of the IDE (BCX). If you are using simulated EMS (such as that provided by QEMM or 386^{MAX}), BCX will also use the simulated EMS; use the **/x** option.

The /h option

If you type `BC/h` or `BCX/h` on the command line, you'll get a list of all the command-line options available. Their default values will also be shown.

The /l option

Use the **/l** option if you're running Borland C++ on an LCD screen.

The /m option

The **/m** option lets you do a make rather than a build (that is, only outdated source files in your project are recompiled and linked). Follow the instructions for the `/b` option, but use `/m` instead.

The /p option

Use the **/p** option, which controls palette swapping on EGA video adapters, when your program modifies the EGA palette registers. The EGA palette will be restored each time the screen is swapped.

In general, you don't need to use this option unless your program modifies the EGA palette registers or unless your program uses BGI to change the palette.

The /r option

Use the **/rx** option if all your extended or expanded memory has been allocated to a RAM disk. The *x* in **/rx** is the letter of the "fast" swap drive. For example, **/rd** will use drive D as the swap drive. This option is primarily for when you have committed all your extended or expanded memory to a RAM disk for other purposes.

Exiting Borland C++

There are three ways to leave the IDE.

- The first method exits the IDE "permanently;" you have to type `BC` or `BCX` again to reenter it. To exit this way, choose **F**ile | **Q**uit (or press *Alt-X*). If you've made changes that you haven't saved, you'll see a prompt asking if you want to save your programs before exiting.
- The next method allows you to shell out from the IDE to enter commands at the DOS command line. To use this method, choose **F**ile | **D**OS Shell. You can enter any normal DOS commands, and you can even run other programs from the command line. When you're ready to return to the IDE, type `EXIT` at the command line and press *Enter*. The IDE reappears just as you left it.

You return to the IDE after you exit the program you transferred to.

- The third method lets you temporarily transfer to another program without leaving the IDE. To do so, choose a program from the ≡ menu. If there are no programs installed on this menu, you can add some with the **O**ptions | **T**ransfer command.

The components

There are three visible components to the IDE: the menu bar at the top, the window area in the middle, and the status line at the bottom. Many menu items also offer dialog boxes. Before we describe each menu item in the IDE, we'll explain these more generic components.

The menu bar and menus

The menu bar is your primary access to all the menu commands. The only time the menu bar is not visible is when you're viewing your program's output or transferring to another program. You'll see a highlighted menu title when the menu bar is active; that menu title is the currently selected menu.

If a menu command is followed by an ellipsis (**...**), choosing the command displays a dialog box. If the command is followed by an arrow (▶), the command leads to another menu (a pop-up menu). If the command has neither an ellipsis nor an arrow, the action occurs as soon as you choose the command.

Here is how you choose menu commands using just the keyboard:

1. Press *F10*. This makes the menu bar active; the next thing you type will relate to the items on the menu bar.
2. Use the arrow keys to select the menu you want to display. Then press *Enter*.

   To cancel an action, press Esc.

   As a shortcut for this step, you can just press the highlighted letter of the menu title. For example, from the menu bar, press *E* to move to and display the **E**dit menu. From anywhere, press *Alt* and the highlighted letter (such as *Alt-E*) to display the menu you want.
3. Use the arrow keys again to select the command you want. Then press *Enter*.

   Again, as a shortcut, you can just press the highlighted letter of a command to choose it once the menu is displayed.

   At this point, Borland C++ either carries out the command, displays a dialog box, or displays another menu.

Borland C++ uses only the left mouse button. You can, however, customize the right button, and make other mouse option changes, by choosing ***Options | Mouse*** *(see page 90).*

You can also use a mouse to choose commands.

1. Click the desired menu title to display the menu.
2. Click the desired command.

Or, drag straight from the menu title down to the menu command. Release the mouse button on the command you want. (If you change your mind, just drag off the menu; no command will be chosen.)

Note that some menu commands are unavailable when it would make no sense to choose them. However, you can still get online help about currently unavailable commands.

Shortcuts

Borland C++ offers a number of quick ways to choose menu commands. (For example, the click-drag method for mouse users.) From the keyboard, you can use a number of keyboard shortcuts (or *hot keys*) to access the menu bar and choose commands. Shortcuts for dialog boxes work just as they do in a menu. (But be aware that you need to hold down *Alt* while pressing the highlighted letter when moving from an input box to a group of buttons or boxes.) Here's a list of the shortcuts available:

Do this...	**To accomplish this...**
Press *Alt* plus the highlighted letter of the command (just press the highlighted letter in a dialog box). For the ≡ menu, press *Alt-Spacebar.*	Display the menu or carry out the command.
Type the keystrokes next to a menu command.	Carry out the command.

For example, to cut selected text, press *Alt-E T* (for **Edit** | **Cut**) or you can just press *Shift-Del,* the shortcut displayed next to it.

Many menu items have corresponding *hot keys;* one- or two-key shortcuts that immediately activate that command or dialog box. The following table lists the most-used Borland C++ hot keys.

Table 1.1
General hot keys

Key(s)	**Menu item**	**Function**
F1	Help	Displays a help screen.
F2	**File** | **Save**	Saves the file that's in the active Edit window.
F3	**File** | **Open**	Brings up a dialog box so you can open a file.
F4	**Run** | **Go to Cursor**	Runs your program to the line where the cursor is positioned.
F5	**Window** | **Zoom**	Zooms the active window.
F6	**Window** | **Next**	Cycles through all open windows.
F7	**Run** | **Trace Into**	Runs your program in debug mode, tracing into functions.
F8	**Run** | **Step Over**	Runs your program in debug mode, stepping over function calls.
F9	**Compile** | **Make EXE File**	Invokes the Project Manager to make an .EXE file.
F10	(none)	Takes you to the menu bar.

Table 1.1: General hot keys (continued)

Table 1.2
Menu hot keys

Key(s)	Menu item	Function
Alt-Spacebar	≡ menu	Takes you to the ≡ (System) menu
Alt-C	**C**ompile menu	Takes you to the Compile menu
Alt-D	**D**ebug menu	Takes you to the Debug menu
Alt-E	**E**dit menu	Takes you to the Edit menu
Alt-F	**F**ile menu	Takes you to the File menu
Alt-H	**H**elp menu	Takes you to the Help menu
Alt-O	**O**ptions menu	Takes you to the Options menu
Alt-P	**P**roject menu	Takes you to the Project menu
Alt-R	**R**un menu	Takes you to the Run menu
Alt-S	**S**earch menu	Takes you to the Search menu
Alt-W	**W**indow menu	Takes you to the Window menu
Alt-X	File | **Q**uit	Exits Borland C++ to DOS

Table 1.3
Editing hot keys

Key(s)	Menu item	Function
Ctrl-Del	Edit | **C**lear	Removes selected text from the window and doesn't put it in the Clipboard
Ctrl-Ins	Edit | **C**opy	Copies selected text to Clipboard
Shift-Del	Edit | **C**ut	Places selected text in the Clipboard, deletes selection
Shift-Ins	Edit | **P**aste	Pastes text from the Clipboard into the active window
Alt-Bkspc	Edit | **U**ndo	Restores the text in the active window to a previous state.
Ctrl-L	Search | **S**earch Again	Repeats last Find or Replace command
F2	File | **S**ave	Saves the file in the active Edit window
F3	File | **O**pen	Lets you open a file

Table 1.3: Editing hot keys (continued)

Table 1.4
Window management hot keys

Key(s)	Menu item	Function
Alt-#		Displays a window, where # is the number of the window you want to view
Alt-0	**W**indow | **L**ist	Displays a list of open windows
Alt-F3	**W**indow | **C**lose	Closes the active window
Alt-F4	**D**ebug | **I**nspect	Opens an Inspector window
Alt-F5	**W**indow | **U**ser Screen	Displays User Screen
F5	**W**indow | **Z**oom	Zooms/unzooms the active window
F6	**W**indow | **N**ext	Switches the active window
Ctrl-F5		Changes size or position of active window

Table 1.5
Online Help hot keys

Key(s)	Menu item	Function
F1	**H**elp | **C**ontents	Opens a context-sensitive help screen
F1 F1		Brings up Help on Help. (Just press *F1* when you're already in the help system.)
Shift-F1	**H**elp | **I**ndex	Brings up Help index
Alt-F1	**H**elp | **P**revious Topic	Displays previous Help screen
Ctrl-F1	**H**elp | **T**opic Search	Calls up language-specific help in Editor only

Table 1.6
Debugging/Running hot keys

Key(s)	Menu item	Function
Alt-F4	**D**ebug | **I**nspect	Opens an Inspector window
Alt-F7	**S**earch | **P**revious Error	Takes you to previous error
Alt-F8	**S**earch | **N**ext Error	Takes you to next error
Alt-F9	**C**ompile | **C**ompile to OBJ	Compiles to .OBJ
Ctrl-F2	**R**un | **P**rogram Reset	Resets running program
Ctrl-F3	**D**ebug | **C**all Stack	Brings up call stack
Ctrl-F4	**D**ebug | **E**valuate/Modify	Evaluates an expression
Ctrl-F7	**D**ebug | **A**dd Watch	Adds a watch expression
Ctrl-F8	**D**ebug | Toggle Breakpoint	Sets or clears conditional breakpoint
Ctrl-F9	**R**un | **R**un	Runs program
F4	**R**un | **G**o To Cursor	Runs program to cursor position
F7	**R**un | **T**race Into	Executes tracing into functions
F8	**R**un | **S**tep Over	Executes skipping function calls
F9	**C**ompile | **M**ake EXE File	Makes (compiles/links) program

Borland C++ windows

Most of what you see and do in the IDE happens in a *window*. A window is a screen area that you can open, close, move, resize, zoom, tile, and overlap.

If you exit Borland C++ with a file open in a window, you are returned to your desktop, open file and all, when you next use Borland C++.

You can have many windows open in the IDE, but only one window can be *active* at any time. The active window is the one that you're currently working in. Any command you choose or text you type generally applies only to the active window. (If you have the same file open in several windows, the action will apply to the file everywhere that it's open.)

You can spot the active window easily: It's the one with the double-lined border around it. The active window always has a close box, a zoom box, and scroll bars. If your windows are overlapping, the active window is always the one on "top" of all the others (the frontmost one).

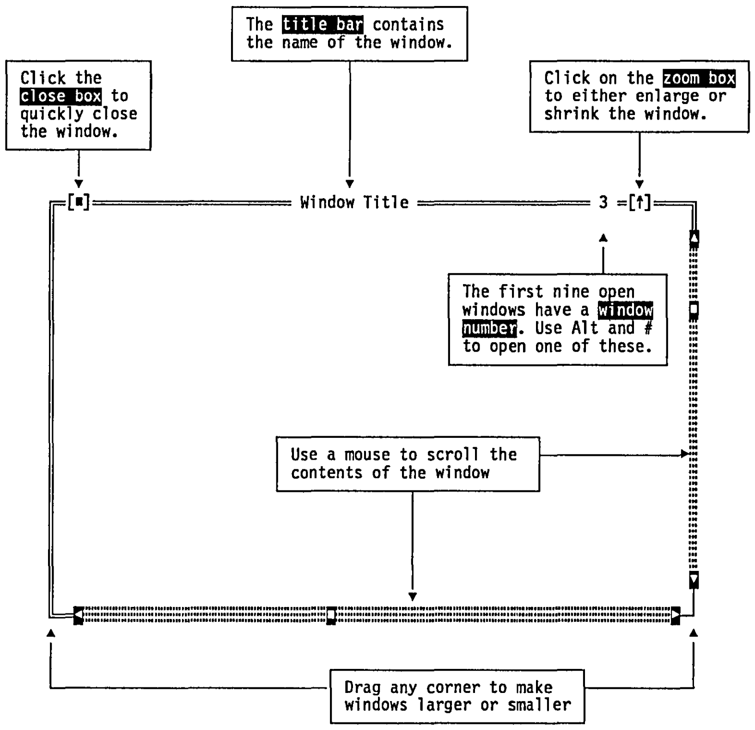
There are several types of windows, but most of them have these things in common:

- a title bar
- a close box
- scroll bars
- a zoom box
- a window number (1 to 9)

The Edit window also displays the current line and column numbers in the lower left corner. If you've modified your file, an asterisk (*) will appear to the left of the column and line numbers.

This is what a typical window looks like:

Figure 1.2
A typical window


The title bar contains the name of the window.
Click the close box to quickly close the window.
Click on the zoom box to either enlarge or shrink the window.
[■]
Window Title
3 =[↑]
The first nine open windows have a window number. Use Alt and # to open one of these.
Use a mouse to scroll the contents of the window
Drag any corner to make windows larger or smaller

Shortcut: Alt-Spacebar invokes the ≡ menu.

The *close box* of a window is the box in the upper left corner. Click this box to quickly close the window. (Or choose **Window | Close** or press *Alt-F3.*) The Inspector and Help windows are considered temporary; you can close them by pressing *Esc.*

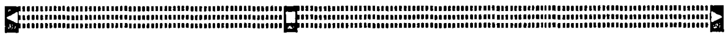
The *title bar,* the topmost horizontal bar of a window, contains the name of the window and the window number. Double-clicking the title bar zooms the window. You can also drag the title bar to move the window around.

***Shortcut:** Double-click the title bar of a window to zoom or restore it.*

The *zoom box* of a window appears in the upper right corner. If the icon in that corner is an up arrow (↑), you can click the arrow to enlarge the window to the largest size possible. If the icon is a doubleheaded arrow (↕), the window is already at its maximum size. In that case, clicking it returns the window to its previous size. To zoom a window from the keyboard, choose **Window | Zoom**, or press *F5.*

The first nine windows you open in Borland C++ have a *window number* in the upper right border. *Alt-0* gives you a list of all windows you have open. You can make a window active (and thereby bring it to the top of the heap) by pressing *Alt* in combination with the window number. For example, if the Help window is #5 but has gotten buried under the other windows, *Alt-5* brings it to the front.

Scroll bars are horizontal or vertical bars that look like this:



Scroll bars also show you where you are in your file.

You use these bars with a mouse to scroll the contents of the window. Click the arrow at either end to scroll one line at a time. (Keep the mouse button pressed to scroll continuously.) You can click the shaded area to either side of the scroll box to scroll a page at a time. Finally, you can drag the scroll box to any spot on the bar to quickly move to a spot in the window relative to the position of the scroll box.

You can drag any corner to make a window larger or smaller. To resize using the keyboard, choose **S**ize/Move from the **W**indow menu, or press *Ctrl-F5*.

Window management

Table 1.7 gives you a quick rundown of how to handle windows in Borland C++. Note that you don't need a mouse to perform these actions—a keyboard works just fine.

Table 1.7
Manipulating windows

To accomplish this:	Use one of these methods
Open an Edit window	Choose **F**ile|**O**pen to open a file and display it in a window, or press *F3*.
Open other windows	Choose the desired window from the **W**indow menu
Close a window	Choose **C**lose from the **W**indow menu (or press *Alt-F3*), or click the close box of the window.
Activate a window	Click anywhere in the window, or Press *Alt* plus the window number (1 to 9, in the upper right border of the window), or

Table 1.7: Manipulating windows (continued)

	Choose **Window** | **List** or press *Alt-0* and select the window from the list, or
	Choose **Window** | **Next** or *F6* to make the next window active (next in the order you first opened them).
Move the active window	Drag its title bar, or press *Ctrl-F5* (**Window** | **Size/Move**) and use the arrow keys to place the window where you want it, then press *Enter.*
Resize the active window	Drag any corner. Or choose **Window** | **Size/Move** and press *Shift* while you use the arrow keys to resize the window, then press *Enter.* The shortcut is to press *Ctrl-F5* and then use *Shift* and the arrow keys.
Zoom the active window	Click the zoom box in the upper right corner of the window, or
	Double-click the window's title bar, or
	Choose **Window** | **Zoom**, or press *F5.*

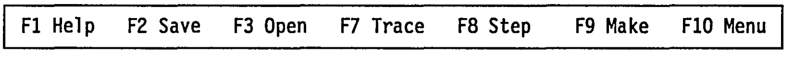
The status line

The status line appears at the bottom of the screen; it

- reminds you of basic keystrokes and shortcuts (or hot keys) applicable at that moment in the active window.
- lets you click the shortcuts to carry out the action instead of choosing the command from the menu or pressing the shortcut keystroke.
- tells you what the program is doing. For example, it displays "Saving *filename...*" when an Edit file is being saved.
- offers one-line hints on any selected menu command and dialog box items.

The status line changes as you switch windows or activities. One of the most common status lines is the one you see when you're actually writing and editing programs in an Edit window. Here is what it looks like:

Figure 1.3
A typical status line



```
F1 Help   F2 Save   F3 Open   F7 Trace   F8 Step   F9 Make   F10 Menu
```

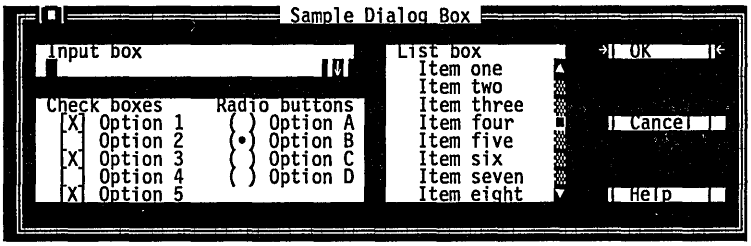
When you've selected a menu title or command, the status line changes to display a one-line summary of the function of the selected item. For example, if the **O**ptions menu title is selected (highlighted), the status line reads "Set defaults for IDE, compiler, debugger; define transfer programs." Similarly, when the **E**dit | Cu**t** command is selected, the status line reads "Remove the selected text and put it in the Clipboard."

Dialog boxes

If a menu command has an ellipsis after it (...), the command opens a *dialog box*. A dialog box is a convenient way to view and set multiple options. When you're making settings in dialog boxes, you work with five basic types of onscreen controls: radio buttons, check boxes, action buttons, input boxes, and list boxes. Here's a sample dialog box that illustrates some of these items:

Figure 1.4
A sample dialog box

If you have a color monitor, Borland C++ uses different colors for various elements of the dialog box.


```
Sample Dialog Box
Input box                          List box          OK
                                     Item one
Check boxes      Radio buttons       Item two
 [X] Option 1     ( ) Option A       Item three       Cancel
 [ ] Option 2     (•) Option B       Item four
 [X] Option 3     ( ) Option C       Item five
 [ ] Option 4     ( ) Option D       Item six
 [X] Option 5                        Item seven
                                     Item eight       Help
```

This dialog box has three standard buttons: OK, Cancel, and Help. If you choose OK, the choices in the dialog box are made; if you choose Cancel, nothing changes and no action is made, but the dialog box is put away. Choose Help to open a Help window about this dialog box. *Esc* is always a keyboard shortcut for Cancel (even if no Cancel button appears).

If you're using a mouse, click the button you want. When you're using the keyboard, press *Alt* and the highlighted letter of an item to activate it. For example, *Alt-K* selects the OK button. Press *Tab* or *Shift-Tab* to move forward or back from one item to another in a dialog box. Each element is highlighted when it becomes active.

In this dialog box, OK is the *default button*, which means you need only press *Enter* to choose that button. (On monochrome systems, arrows indicate the default; on color monitors, default buttons are highlighted.) You can select another button with *Tab;* press *Enter* to choose that button. Be aware that tabbing to a button makes that button the default.

Check boxes and radio buttons

```
[X] Checked check box
[ ] Unchecked check box
```

The dialog box also has *check boxes.* When you select a check box, an *x* appears in it to show you it's on. An empty box indicates it's off. To set a check box to on, click it or its text, by pressing *Tab* until the check box is highlighted and then pressing *Spacebar,* or by selecting *Alt* and the highlighted letter. You can have any number of check boxes checked at any time.

If several check boxes apply to a topic, they appear as a group. In that case, tabbing moves to the group. Once the group is selected, use the arrow keys to select the item you want, and then press *Spacebar* to choose it. On monochrome monitors, the active check box or group of check boxes will have a chevron symbol (») to the left and right. When you press *Tab,* the chevrons move to the next group of checkboxes or radio buttons.

The dialog box also has *radio buttons.* Radio buttons are so called because they act just like the buttons on a car radio. There is always one—and only one—button pushed in at a time. Push one in, and the one that was in pops out.

```
( ) None
(•) Emulation
( ) 8087
```

Radio buttons differ from check boxes in that they present mutually exclusive choices. For this reason, radio buttons always come in groups, and exactly one (no more, no less) radio button can be on in any one group at any one time. To choose a radio button, click it or its text. From the keyboard, select *Alt* and the highlighted letter, or press *Tab* until the group is highlighted and then use the arrow keys to choose a particular radio button. Press *Tab* or *Shift-Tab* again to leave the group with the new radio button chosen. The column to the left gives an example a a set of radio buttons.

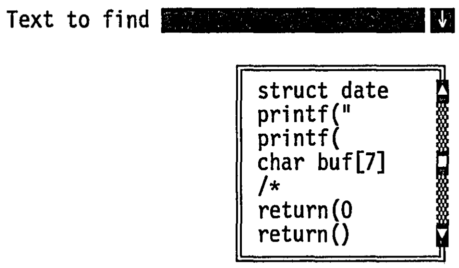
Input boxes and lists

Dialog boxes can also contain input boxes, which allow you to type in text. Most basic text-editing keys work in the text box (for example, arrow keys, *Home, End,* and insert/overwrite toggles by *Ins*). If you continue to type once you reach the end of the box, the contents automatically scroll. If there's more text than what shows in the box, arrowheads appear at the end (◂ and ▸). You can click the arrowheads to scroll or drag the text. If you need to enter control characters (such as ^L or ^M) in the input box, then prefix the character with a ^P. So, for example, entering ^P^L enters a ^L into the input box. (This ability is useful for search strings.)

*You can control whether history lists are saved to the desktop using **O**ptions | **E**nvironment | **D**esktop.*

If an input box has a down-arrow icon to its right, there is a *history list* associated with that input box. Press *Enter* to select an item from this list. In the list you'll find text you typed into this box the last few times you used this dialog box. The Find box, for example, has such a history list, which keeps track of the text you searched for previously. If you want to reenter text that you already entered, press ↓ or click the ↓ icon. You can also edit an entry in the history list. Press *Esc* to exit from the history list without making a selection.

Here is what a history list for the Find text box might look like if you had used it seven times previously:



```
Text to find
struct date
printf("
printf(
char buf[7]
/*
return(0
return()
```

A final component of many dialog boxes is a *list box.* A list box lets you scroll through and select from variable-length lists (often file names) without leaving a dialog box. If a blinking cursor appears in the list box and you know what you're looking for, you can type the word (or the first few letters of the word) and Borland C++ will search for it.

You make a list box active by clicking it or by choosing the highlighted letter of the list title (or press *Tab* until it's highlighted). Once a list box is displayed, you can use the scroll box to move through the list or press ↑ or ↓ from the keyboard.

Editing

If you're a longtime user of Borland products, the following summary of our major editing features can help you identify the areas that are different from our older products.

- multi-file capabilities let you open several files at once

- multiple windows let you have several views onto the same file or different files
- block mode that is switchable between persistent and destructive (see page 93)
- mouse support
- support for large files (greater than 64K; limited to 8 megabytes for all edit windows combined)
- *Shift* ↑ ↓ → ← for selecting text
- edit windows that you can move, resize, or overlap
- a sophisticated macro language, so you can create your own editor commands (documented online)
- a built-in assembler and support for inline assembler code
- an undo and redo feature with an extensive buffer
- the ability to paste examples from the Help window
- an editable Clipboard that allows cutting, copying, and pasting in or between windows
- a Transfer function that lets you run other programs and capture output to an editor without leaving Borland C++

Project and configuration files

The IDE handles configuration files differently than Turbo C. The focus of the IDE has changed from configuration-based to project-based. This means that instead of loading a configuration (.TC) file that defines your project, you load a project file that contains everything needed to build your program.

Turbo C files

In Turbo C, all options (compiler, environment, and so on) are stored in the .TC file. The project file consists of an ASCII list of file names that comprise the project. Thus, the information needed to build the program that the project represents is spread across two files: the project file and the .TC file.

Borland C++ project files

The IDE places all information needed to build a program into a binary project file. This includes compiler and linker options, directory paths, project specific settings (for example, program heap size, autodependencies used, and so on), and special translators (such as TASM). In addition, the project file contains other general information on the project, such as compilation statistics (shown in the project window), and cached autodependency information. Chapter 4, "Managing multi-file projects," covers project files and the Project Manager in depth.

Configuration files

The TCCONFIG.TC file contains only environmental (or global) information; project files (.PRJ) now contain information on all other settings and options except those shown in the following list. For instance, the TCCONFIG.TC file knows whether to save breakpoints and watchpoints, but, if activated, the breakpoints and watchpoints themselves will be saved in the .PRJ. Thus the .TC file is no longer required to build programs defined by a project. The information stored in the .TC file includes

You might need only one copy of the .TC file on your hard disk. When kept with the BC.EXE file, the TCCONFIG.TC file is loaded, unless there is another copy in the current directory.

- editor key binding and macros
- editor mode setting (such as autoindent, use tabs, etc.)
- color tables
- 25/43 line setting
- mouse preferences
- auto-save flags
- history lists

Project files for the IDE correspond to the .CFG configuration files that you supply to the command-line compiler (the default command-line compiler configuration file is TURBOC.CFG). The PRJCFG utility can convert .PRJ files to .CFG files and .CFG files to .PRJ files. See Chapter 7, page 189 for more information.

Loading project files

You can load project files in any of three ways:

1. When invoking Borland C++, give the project name with the .PRJ extension after the *BC* command; for example,

   ```
   BC myproj.PRJ
   ```

   You must use the .PRJ extension in order to differentiate it from source files.
2. If there is only one .PRJ file in the current directory, the IDE assumes that this directory is dedicated to this project and automatically loads it. Thus, typing `BC` (or `BCX`) alone while the current directory contains one project file causes that project file to be loaded.
3. From within the IDE, you load a project file using the **P**roject | **O**pen Project command.

The project directory

When a project file is loaded from a directory other than the current directory, the current DOS directory is set to where the project is loaded from. This allows your project to be defined in terms of relative paths in the **O**ptions | **D**irectories dialog box and also allows projects to move from one drive to another or from one directory branch to another. Note, however, that changing directories after loading a project may make the relative paths incorrect and your project unbuildable. If this happens, change the current directory back to where the project was loaded from.

Desktop files

Each project file has an associated desktop file (*prjname*.DSK). This file contains state information about the associated project. While none of the information it contains is needed to build the project, all of the information is directly related to the project. The desktop file includes

*You can set some of these options on or off using **O**ptions | **E**nvironment | **D**esktop.*

- the context information for each file in the project (that is, the position in the file, the location of the window on the screen, and so on)
- the history lists for various input boxes (for example, search strings, file masks, and so on)
- layout of the windows on the desktop
- the contents of the Clipboard
- watch expressions
- breakpoints

Changing project files

Because each project file has its own desktop file, changing to another project file causes the newly loaded project's desktop to be used. Thus changing from one existing project to another existing project can change your entire window layout. When you create a new project (by using **P**roject | **O**pen Project and typing in a new .PRJ file), the new project's desktop inherits the previous desktop. When you select **P**roject | **C**lose Project, the default project is loaded and you get the default desktop and project settings.

Default files

When no project file is loaded, there are two default files that serve as global place holders for project- and state-related information: TCDEF.DPR and TCDEF.DSK files, collectively referred to as the default project.

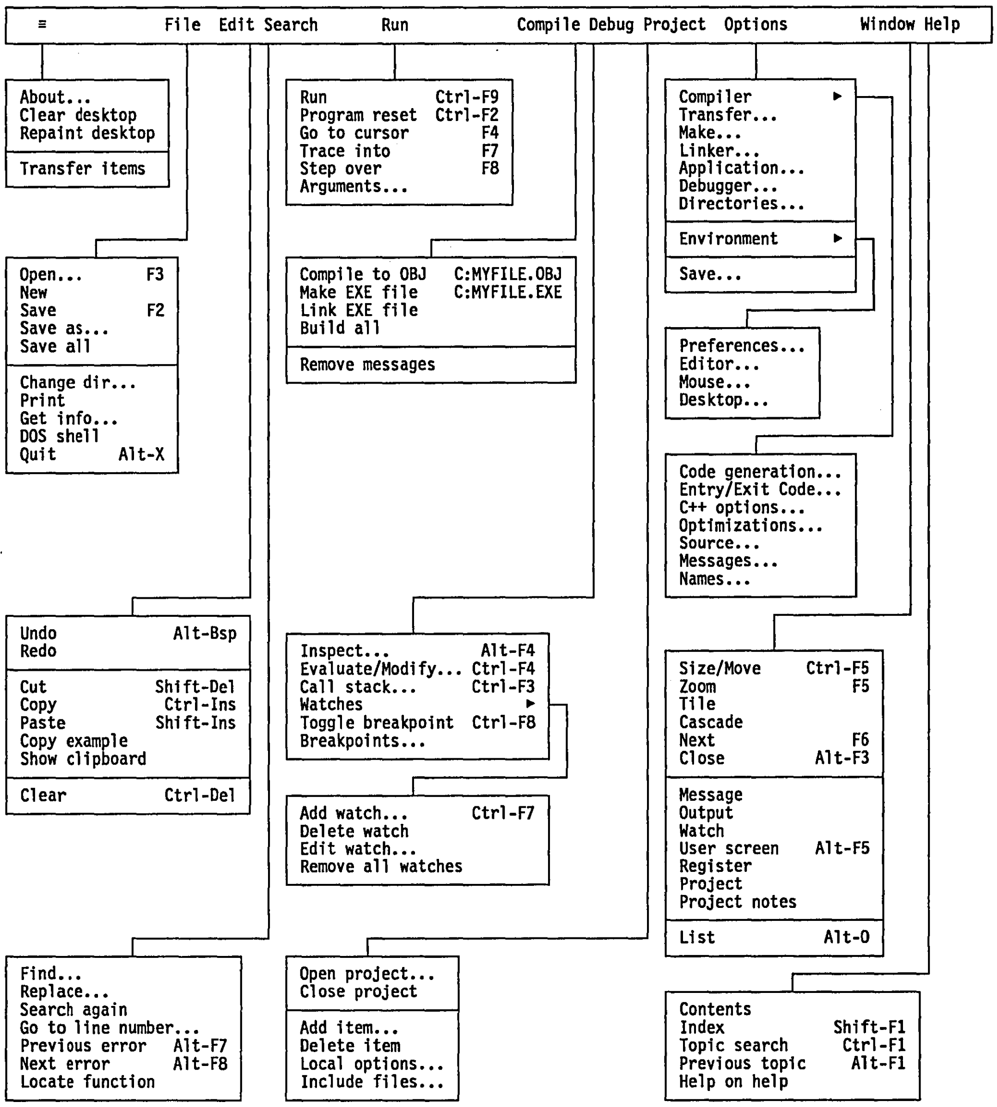
These files are usually stored in the same directory as BC.EXE, and are created if they are not found. When you run the IDE from a directory without loading a project file, you get the desktop and settings from these files. These files are updated when you change any project-related options (for example, compiler options) or when your desktop changes (for example, the window layout).

C H A P T E R

2

Menus and options reference

This chapter provides a reference to each menu item and dialog box. It is arranged in the order that the menus appear on the screen. For information on starting and exiting the IDE, using the IDE command-line options, and general information on how the IDE works, see Chapter 1. Here is a high-level summary of the menus:



≡ (System) menu

Alt Space

The ≡ menu appears on the far left of the menu bar. *Alt-Spacebar* is the fastest way to get there. When you pull down this menu, you see several general system-wide commands (**A**bout, **C**lear

Desktop, **R**epaint Desktop) and the names of programs you've installed with the **O**ptions | **T**ransfer command.

About

The first command in the menu is **A**bout. When you choose this command, a dialog box appears that shows you copyright and version information for Borland C++. Press *Esc* or click OK (or press *Enter*) to close the box.

Clear Desktop

Choose ≡ | **C**lear Desktop to close all windows and clear all history lists. This command is useful when you're starting a new project.

Repaint Desktop

Choose ≡ | **R**epaint Desktop to have Borland C++ redraw the screen. You may need to do this, for example, if a memory-resident program has left stray characters on the screen, or possibly if you have screen-swapping turned off (**O**ptions | **D**ebug | Display swapping) and you're stepping through a program.

Transfer items

Any programs you've installed with the Transfer dialog box (**O**ptions | **T**ransfer) appear here. To run one of these programs, choose its name from the ≡ menu. To install programs that will then appear in this menu, choose **O**ptions | **T**ransfer.

If you have more than one program installed with the same shortcut letter on this menu, the first program listed with that shortcut will be selected. You can select the second item by clicking it or by using the arrow keys to move to it and then pressing *Enter.*

File menu

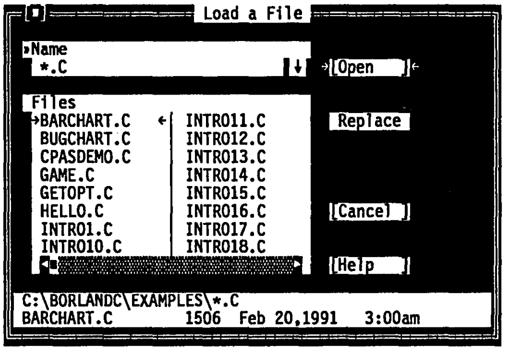
[Alt][F] The **F**ile menu lets you open and create program files in Edit windows. The menu also lets you save your changes, perform other file functions, shell to DOS, and quit.

Open

F3

The **File | Open** command displays a file-selection dialog box for you to select a program file to open in an Edit window. Here is what the box looks like:

Figure 2.1
The Load a File dialog box



```
Name
 *.C

Files
BARCHART.C      INTRO11.C
BUGCHART.C      INTRO12.C
CPASDEMO.C      INTRO13.C
GAME.C          INTRO14.C
GETOPT.C        INTRO15.C
HELLO.C         INTRO16.C
INTRO1.C        INTRO17.C
INTRO10.C       INTRO18.C

C:\BORLANDC\EXAMPLES\*.C
BARCHART.C      1506   Feb 20,1991    3:00am
```

The dialog box contains an input box, a file list, buttons labeled Open, Replace, Cancel, and Help, and an information panel that describes the selected file. Now you can do any of these actions:

- Type in a full file name and choose Replace or Open. Open loads the file into a new Edit window. An Edit window must be active if you choose Replace; the contents of the window is replaced with the selected file.
- Type in a file name with wildcards, which filters the file list to match your specifications.
- Press ↓ to choose a file specification from a history list of file specifications you've entered earlier.
- View the contents of different directories by selecting a directory name in the file list.

The input box lets you enter a file name explicitly or lets you enter a file name with standard DOS wildcards (* and ?) to filter the names appearing in the history list box. If you enter the entire name and press *Enter*, Borland C++ opens it. (If you enter a file name that Borland C++ can't find, it automatically creates and opens a new file with that name.)

If you press ↓ when the cursor is blinking in the input box, a history list drops down below the box. This list displays the last eight file names you've entered. Choose a name from the list by double-clicking it or selecting it with the arrow keys and pressing *Enter*.

If you choose Replace instead of Open, the selected file replaces the file in the active Edit window instead of opening up a new window.

Once you've typed in or selected the file you want, choose the Open button (choose Cancel if you change your mind). You can also just press *Enter* once the file is selected, or you can double-click the file name.

Using the File list box

You can also type a lowercase letter to search for a file name and an uppercase letter to search for a directory name.

The File list box displays all file names in the current directory that match the specifications in the input box, displays the parent directory, and displays all subdirectories. Click the list box or press *Tab* until the list box name is highlighted. You can now press ↓ or ↑ to select a file name, and then press *Enter* to open it. You can also double-click any file name in the box to open it. You might have to scroll the box to see all the names. If you have more than one pane of names, you can also use → and ←.

The file information panel at the bottom of the Load a File dialog box displays path name, file name, date, time, and size of the file you've selected in the list box. (None of the items on this panel are selectable.) As you scroll through the list box, the panel is updated for each file.

New

The **File | New** command lets you open a new Edit window with the default name NONAME*xx*.C (the *xx* stands for a number from 00 to 99). These NONAME files are used as a temporary edit buffer; Borland C++ prompts you to name a NONAME file when you save it.

Save

F2

The **File | Save** command saves the file in the active Edit window to disk. (This menu item is disabled if there's no active Edit window.) If the file has a default name (NONAME00.C, or the like), Borland C++ opens the Save Editor File dialog box to let you rename and save it in a different directory or on a different drive. This dialog box is identical to the one opened for the **Save** As command, described next.

Save As

The **F**ile | **S**ave **A**s command lets you save the file in the active Edit window under a different name, in a different directory, or on a different drive. When you choose this command, you see the Save File As dialog box:

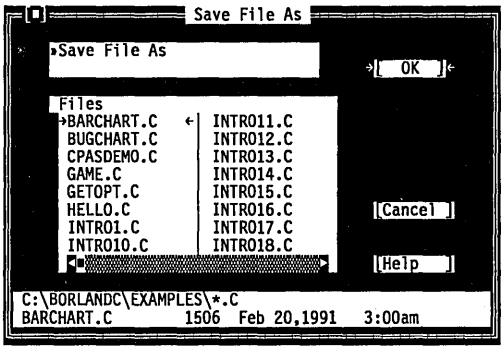


Figure 2.2
The Save File As dialog box

```
Save File As
Save File As
Files
BARCHART.C      INTRO11.C
BUGCHART.C      INTRO12.C
CPASDEMO.C      INTRO13.C
GAME.C          INTRO14.C
GETOPT.C        INTRO15.C
HELLO.C         INTRO16.C
INTRO1.C        INTRO17.C
INTRO10.C       INTRO18.C
C:\BORLANDC\EXAMPLES\*.C
BARCHART.C        1506  Feb 20,1991   3:00am
```

Enter the new name, optionally with drive and directory, and click or choose OK. All windows containing this file are updated with the new name.

Save All

The **F**ile | Save **A**ll command works just like the **S**ave command except that it saves the contents of all modified files, not just the file in the active Edit window. This command is disabled if no Edit windows are open.

Change Dir

The **F**ile | **C**hange Dir command lets you specify a drive and a directory to make current. The current directory is the one Borland C++ uses to save files and to look for files. (When using relative paths in **O**ptions | **D**irectories, they are relative to this current directory only.)

Here is what the Change Directory dialog box looks like:

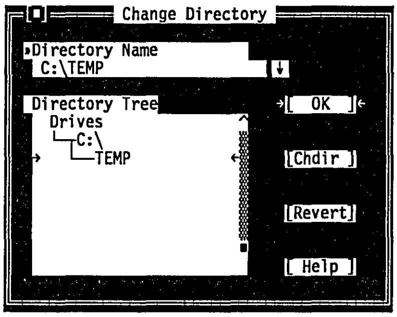


Figure 2.3
The Change Directory dialog box

There are two ways to change directories:

- Type in the path of the new directory in the input box and press *Enter*, or
- Choose the directory you want in the Directory tree (if you're using the keyboard, press *Enter* to make it the current directory), then choose OK or press *Esc* to exit the dialog box.

If you choose the OK button, your changes will be made and the dialog box put away. If you choose the Chdir button, the Directory Tree list box changes to the selected directory and displays the subdirectories of the currently highlighted directory (pressing *Enter* or double-clicking on that entry gives you the same result). If you change your mind about the directory you've picked and you want to go back to the previous one (*and* you've yet to exit the dialog box), choose the Revert button.

Print

The **File | P**rint command lets you print the contents of the active Edit, Output, or Message window. Borland C++ expands tabs (replaces tab characters with the appropriate number of spaces) and then sends it to the DOS print handler. This command is disabled if the active window cannot be printed. Use *Ctrl-K P* to print selected text only.

Get Info

The **File | G**et Info command displays a box with information on the current file.

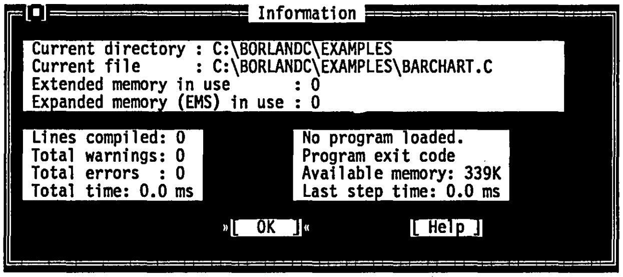


Figure 2.4
The Get Info box

```
Information

Current directory : C:\BORLANDC\EXAMPLES
Current file      : C:\BORLANDC\EXAMPLES\BARCHART.C
Extended memory in use      : 0
Expanded memory (EMS) in use : 0

Lines compiled: 0          No program loaded.
Total warnings: 0          Program exit code
Total errors  : 0          Available memory: 339K
Total time: 0.0 ms         Last step time: 0.0 ms
```

The information here is for display only; you can't change any of the settings in this box. The following table tells you what each line in the Get Info box means and where you can go to change the settings if you want to:

Table 2.1
Get Info settings

After reviewing the information in this box, press Enter to put the box away.

Setting	Meaning
Current directory	The default directory.
Current file	File in the active window.
Extended memory in use	Amount of extended memory reserved by Borland C++; displays 0 when the IDE is in protected mode.
Expanded memory in use	Amount of expanded memory reserved by Borland C++.
Lines compiled	Number of lines compiled.
Total warnings	Number of warnings issued.
Total errors	Number of errors generated.
Total time	Amount of time your program has run (debugger only).
Program loaded	Debugging status.
Program exit code	DOS termination code of last terminated program.
Available memory	Amount of free DOS (640K) memory. In protected mode, this value is the number of bytes of extended memory.
Last step time	Amount of time spent in last debug step.

DOS Shell

The **File | DOS** Shell command lets you temporarily exit Borland C++ to enter a DOS command or program. To return to Borland C++, type `EXIT` and press *Enter.*

You may find that when you're debugging, there's not enough memory to execute this command. If that's the case, terminate the debug session by choosing **Run | P**rogram Reset (*Ctrl-F2*).

Don't install any TSR programs (like SideKick) while you've shelled to DOS, because memory may get misallocated.

Note: In dual monitor mode, the DOS command line appears on the Borland C++ screen rather than the User Screen. This allows you to switch to DOS without disturbing the output of your program. Since your program output is available on one monitor in the system, **W**indow | **U**ser Screen and *Alt-F5* are disabled.

You can also use the transfer items on the ≡ (System) menu to quickly switch to another program without leaving Borland C++.

Quit

Alt X

The **File** | **Quit** command exits Borland C++, removes it from memory, and returns you to the DOS command line. If you have made any changes that you haven't saved, Borland C++ asks you if you want to save them before exiting.

Edit menu

Alt E

The **Edit** menu lets you cut, copy, and paste text in Edit windows. If you make mistakes, you can undo changes, and even reverse the effect of your most recent undo. You can also open a Clipboard window to view or edit its contents, and copy text from the Message and Output windows.

Before you can use most of the commands on this menu, you need to know about selecting text (because most editor actions apply to selected text). Selecting text means highlighting it. You can select text either with keyboard commands or with a mouse; the principle is the same even though the actions are different.

From the keyboard you can use any of these methods:

- Press *Shift* while pressing any arrow key.
- To select text from the keyboard, press *Ctrl-K B* to mark the start of the block. Then move the cursor to the end of the text and press *Ctrl-K K*.
- To select a single word, move the cursor to the word and press *Ctrl-K T*.
- To select an entire line, press *Ctrl-K L*.

With a mouse:

- To select text with a mouse, drag the mouse pointer over the desired text. If you need to continue the selection past a window's edge, just drag off the side and the window will automatically scroll.
- To select a single line, double-click anywhere in the line.
- To select text line-by-line, click-drag over the text (that is, click once and then quickly press the mouse button again and begin to drag).
- To extend or reduce the selection, Shift-click anywhere in the document (that is, hold *Shift* and click).

Once you have selected text, the commands in the **E**dit menu become available, and the Clipboard becomes useful.

The Clipboard is the magic behind cutting and pasting. It's a special window in Borland C++ that holds text that you have cut or copied, so you can paste it elsewhere. The Clipboard works in close concert with the commands in the **E**dit menu.

Here's an explanation of each command in the **E**dit menu.

Undo

Alt Backspace

The **E**dit | **U**ndo command restores the file in the current window to the way it was before the most-recent edit or cursor movement. There are restrictions on what operations can be undone. Undo will insert any characters that have been deleted, delete any characters that have been inserted, replace any characters that have been overwritten, and move the cursor back to a prior position.

If you delete large blocks, you may lose your Undo information.

Undoing a block operation restores the block markers to the value they had prior to the operation. Undo will not change any flag option setting that has a global effect. For example, if you use the *Ins* key to change from Insert to Overwrite mode, then choose **U**ndo, you won't change back into Insert mode. However, if you delete a character, switch to Overwrite mode, then choose **U**ndo, the character you just deleted will be inserted. The effect of the operation that you performed (deleting a character) is undone regardless of the mode setting.

Changing global editing options may make Undo act differently than you expect. For instance, if you press *Tab*, then change the value of Tab Width in the Editor Options dialog box, then choose Undo, the cursor will move to the old tab location (based on certain values in the Undo buffer). As soon as you start typing, however, the cursor will move to the new tab location.

The Group Undo option in the Editor Options dialog box (**Options | Environment | Editor**) affects **Undo** and **Redo**. See page 93 for information on Group Undo.

Redo

The **Edit | Redo** command reverses the effect of the most recent **Undo** command. The **Redo** command only has an effect immediately after an **Undo** command or after another **Redo** command. A series of **Redo** commands reverses the effects of a series of **Undo** commands.

Cut

Shift Del

The **Edit | Cut** command removes the selected text from your document and places the text in the Clipboard. You can then paste that text into any other document (or somewhere else in the same document) by choosing **Paste**. The text remains selected in the Clipboard so that you can paste the same text many times.

Copy

Ctrl Ins

The **Edit | Copy** command leaves the selected text intact but places an exact copy of it in the Clipboard. You can then paste that text into any other document by choosing **Paste**. You can also copy text from a Help window: With the keyboard, use *Shift* and the arrow keys; with the mouse, click and drag the text you want to copy. If the Output or Message window is the active window when you select **Edit | Copy**, the entire contents of the window buffer (including any nonvisible portion) get copied to the Clipboard.

Paste

Shift Ins

The **Edit | Paste** command inserts text from the Clipboard into the current window at the cursor position. The text that is actually pasted is the currently marked block in the Clipboard window. You cannot paste to either the Output or Message window.

Copy Example

The **E**dit | Copy **E**xample command copies the preselected example text in the current Help window to the Clipboard. The examples are already predefined as pastable blocks, so you don't need to bother with marking the example you want.

Show Clipboard

The **E**dit | **S**how Clipboard command opens the Clipboard window, which stores the text you cut and copy from other windows. The text that's currently selected (highlighted) is the text Borland C++ uses when you choose **P**aste.

You can think of the Clipboard window as a history list of your cuts and copies. And you can edit the Clipboard so that the text you paste is precisely the text you want. Borland C++ uses whatever text is selected in the Clipboard when you choose **P**aste.

*Normally the Clipboard contents are saved across sessions, although you can control that using **O**ptions | **E**nvironment | **D**esktop.*

The Clipboard window is just like other Edit windows; you can move it, resize it, and scroll and edit its contents. The only difference you'll find in the Clipboard window is when you choose to cut or copy text. When you select text in the Clipboard window and choose Cu**t** or **C**opy, the selected text immediately appears at the bottom of the window. (Remember, any text that you cut or copy is appended to the end of the Clipboard—so you can paste it later.)

Clear

[Ctrl][Del]

The **E**dit | Clear command removes the selected text but does not put it into the Clipboard. This means you cannot paste the text as you could if you had chosen Cu**t** or **C**opy. The cleared text is not retrievable. You can clear the Clipboard itself by selecting all the text in the Clipboard, then selecting **E**dit | **C**lear.

Search menu

[Alt][S]

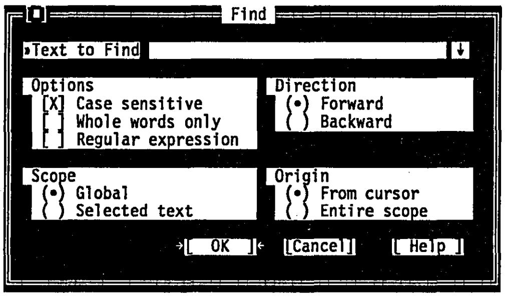
The **S**earch menu lets you search for text, function declarations, and error locations in your files.

Find

Alt S F

The **Search | Find** command displays the Find dialog box, which lets you type in the text you want to search for and set options that affect the search. (*Ctrl-Q F* is another shortcut for this command.)

Figure 2.5
The Find dialog box



The Find dialog box contains several buttons and check boxes:

`[X] Case sensitive`

Check the Case Sensitive box if you do want Borland C++ to differentiate uppercase from lowercase.

`[ ] Whole words only`

Check the Whole Words Only box if you want Borland C++ to search for words only (that is, the string must have punctuation or space characters on both sides).

`[ ] Regular expression`

Check the Regular Expression box if you want Borland C++ to recognize GREP-like wildcards in the search string. The wildcards are ^, $, ., *, +, [], and \. Here's what they mean:

^	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.
*	A character followed by an asterisk matches any number of occurrences (including zero) of that character. For example, *bo** matches *bot*, *b*, *boo*, and also *be*.
+	A character followed by a plus sign matches any number of occurrences (but not zero) of that character. For example, *bo+* matches *bot* and *boo*, but not *be* or *b*.

[ ]	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 Borland C++ to treat that character literally, not as a wildcard. For example, \^ matches ^ and does not look for the start of a line.

Enter the string in the input box and choose OK to begin the search, or choose Cancel to forget it. If you want to enter a string that you searched for previously, press ↓ to show a history list to choose from.

You can also pick up the word that your cursor is currently on in the Edit window and use it in the Find box by simply invoking **F**ind from the **S**earch menu. You can take additional characters from the text by pressing →.

```
Direction
 (•) Forward
 ( ) Backward
```

Choose from the Direction radio buttons to decide which direction you want Borland C++ to search—starting from the origin (which you can set with the Origin radio buttons).

```
Scope
 (•) Global
 ( ) Selected text
```

Choose from the Scope buttons to determine how much of the file to search in. You can search the entire file (Global) or only the selected text.

```
Origin
 (•) From Cursor
 ( ) Entire Scope
```

Choose from the Origin buttons to determine where the search begins. When Entire Scope is chosen, the Direction radio buttons determine whether the search starts at the beginning or the end of the scope. You choose the range of scope you want with the Scope radio buttons.

Replace

Ctrl Q A

The **S**earch | **R**eplace command displays a dialog box that lets you type in the text you want to search for and text you want to replace it with.

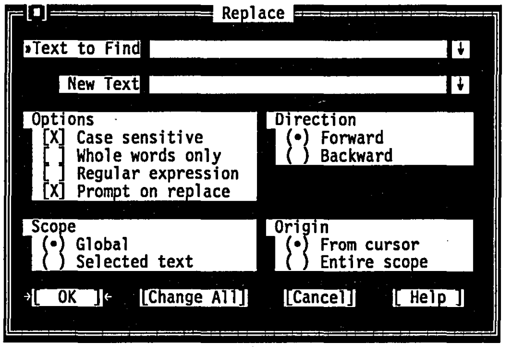


Figure 2.6
The Replace dialog box

The Replace dialog box contains several radio buttons and check boxes—many of which are identical to the Find dialog box, discussed previously. An additional checkbox, Prompt on Replace, controls whether you're prompted for each change.

Enter the search string and the replacement string in the input boxes and choose OK or Change All to begin the search, or choose Cancel to forget it. If you want to enter a string you used previously, press ↓ to show a history list to choose from.

If Borland C++ finds the specified text, it asks you if you want to make the replacement. If you choose OK, it will find and replace only the first instance of the search item. If you choose Change All, it replaces all occurrences found, as defined by Direction, Scope, and Origin.

As with the Find dialog box, you can pick up the word your cursor is currently on in the Edit window and use it in the Text to Find input box by simply invoking **F**ind or **R**eplace from the **S**earch menu. And you can add more text from the Edit window by pressing →.

Search Again

Ctrl L

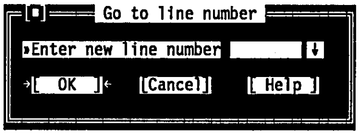
The **S**earch | **S**earch Again command repeats the last **F**ind or **R**eplace command. All settings you made in the last dialog box used (Find or Replace) remain in effect when you choose **S**earch Again.

Go to Line Number

The **S**earch | **G**o to Line Number command prompts you for the line number you want to find.

Here is what the dialog box looks like:

Figure 2.7
The Go to Line Number dialog box



Borland C++ displays the current line number and column number in the lower left corner of every Edit window.

Previous Error

Alt F7

The **S**earch | **P**revious Error command moves the cursor to the location of the previous error or warning message. This command is available only if there are messages in the Message window that have associated line numbers. These messages are generated by compile and transfer commands that use a Capture messages filter.

Next Error

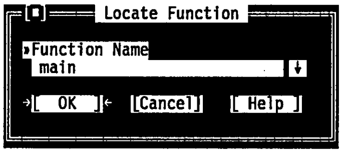
Alt F8

The **S**earch | **N**ext Error command moves the cursor to the location of the next error or warning message. This command is available only if there are messages in the Message window that have associated line numbers. These messages are generated by compile and transfer commands that use a Capture messages filter.

Locate Function

The **S**earch | **L**ocate Function command displays a dialog box for you to enter the name of a function to search for. This command is available only during a debugging session.

Figure 2.8
The Locate Function dialog box



Enter the name of a function or press ↓ to choose a name from the history list. As opposed to the **F**ind command, this command finds the declaration of the function, not instances of its use.

Run menu

Alt R

The **Run** menu's commands run your program, and also start and end debugging sessions.

Run

Ctrl F9

The **Run | Run** command runs your program, using any arguments you pass to it with the **Run | Arguments** command. If the source code has been modified since the last compilation, it will also invoke the Project Manager to recompile and link your program. (The Project Manager is a program building tool incorporated into the IDE; see Chapter 4, "Managing multi-file projects," for more on this feature.)

If you want to have all Borland C++'s features available, keep Source Debugging set to On.

If you don't want to debug your program, you can compile and link it with the Source Debugging radio button set to None (which makes your program compile and link faster) or to Standalone (which gives the program more room to run) in the **O**ptions | De**b**ugger dialog box. If you compile your program with this check box set to On, the resulting executable code will contain debugging information that will affect the behavior of the **Run | Run** command in the following ways:

Source code the same

If you have not modified your source code since the last compilation,

- the **Run | Run** command causes your program to run to the next breakpoint, or to the end if no breakpoints have been set.

Source code modified

If you have modified your source code since the last compilation,

- and if you're already stepping through your program using the **Run | Step** Over or **Run | Trace** Into commands, **Run | Run** prompts you whether you want to rebuild your program:
  - If you answer yes, the Project Manager recompiles and links your program, and sets it to run from the beginning.
  - If you answer no, your program runs to the next breakpoint or to the end if no breakpoints are set.
- and if you are not in an active debugging session, the Project Manager recompiles your program and sets it to run from the beginning.

Pressing *Ctrl-Break* causes Borland C++ to stop execution on the next source line in your program. If Borland C++ is unable to find a source line, a second *Ctrl-Break* will terminate the program and return you to the IDE.

Windows You can't run or debug Windows applications within the IDE. If you try to do so, you'll get an error dialog box to that effect.

Program Reset

Ctrl F2

The **Run | P**rogram Reset command stops the current debugging session, releases memory your program has allocated, and closes any open files that your program was using. Use this command when you're debugging and there's not enough memory to run transfer programs or invoke a DOS shell.

Go to Cursor

F4

The **Run | G**o to Cursor command runs your program from the run bar (the highlighted bar in your code) to the line the cursor is on in the current Edit window. If the cursor is at a line that does not contain an executable statement, the command displays a warning. **Run | G**o to Cursor can also initiate a debug session.

Go to Cursor does not set a permanent breakpoint, but it does allow the program to stop at a permanent breakpoint if it encounters one before the line the cursor is on. If this occurs, you must choose the **G**o to Cursor command again.

Use **G**o to Cursor to advance the run bar to the part of your program you want to debug. If you want your program to stop at a certain statement every time it reaches that point, set a breakpoint on that line.

Note that if you position the cursor on a line of code that is not executed, your program will run to the next breakpoint or the end if no breakpoints are encountered. You can always use *Ctrl-Break* to stop a running program.

Trace Into

F7

The **Run | T**race Into command runs your program statement-by-statement. When it reaches a function call, it executes each statement within the function, instead of executing the function as a single step (see **Run | S**tep Over). If a statement contains no calls to functions accessible to the debugger, **T**race Into stops at the next executable statement.

Use the **T**race Into command to move the run position into a function called by the function you are now debugging. See the next section for an illustration of the differences between the **T**race Into and **S**tep Over commands.

If the statement contains a call to a function accessible to the debugger, **T**race Into halts at the beginning of the function's definition. Subsequent **T**race Into or **S**tep Over commands run the statements in the function's definition. When the debugger leaves the function, it resumes evaluating the statement that contains the call; for example,

```
if (func1() && func2())
   do_something();
```

With the run bar on the **if** statement, *F7* will trace into **func1**; when on the return in **func1**, *F7* will trace into **func2**. *F8* will step over **func2** and stop on *do-something*.

Note: The **T**race Into command recognizes only functions defined in a source file compiled with two options set on:

- In the **C**ode Generation dialog box (**O**ptions | **C**ompiler), the Debug Info in OBJs check box must be checked.
- The Source Debugging radio buttons must be set to On (in the **O**ptions | De**b**ugger dialog box).

Step Over

F8

The **Run** | **S**tep Over command executes the next statement in the current function. It does not trace into calls to lower-level functions, even if they are accessible to the debugger.

Use **S**tep Over to run the function you are now debugging, one statement at a time without branching off into other functions.

Here is an example of the difference between **R**un | **T**race Into and **R**un | **S**tep Over. These are the first 12 lines of a program loaded into an Edit window:

```
int findit(void)        /* Line 1 */
{
   return(2);
}
```

```
void main(void)         /* Line 6 */
{
   int i, j;

   i = findit();        /* Line 10 */
   printf("%d\n", i);   /* Line 11 */
   j = 0; . . .         /* Line 12 */
```

findit is a user-defined function in a module that has been compiled with debugging information. Suppose the run bar is on line 10 of your program. To position the run bar on line 10, place the cursor on line 10 and either press *F4* or select **Run | Go** to Cursor.

- If you now choose **Run | T**race Into, the run bar will move to the first line of the **findit** function (line 1 of your program), allowing you to step through the function.
- If you choose **Run | S**tep Over, the **findit** function will execute and the return value will be assigned to *i*. Then the run bar will move to line 11.

If the run bar had been on line 11 of your program, it would have made no difference which command you chose; **Run | T**race Into and **Run | S**tep Over both would have executed the **printf** function and moved the run bar to line 12. This is because the **printf** function does not contain debug information.

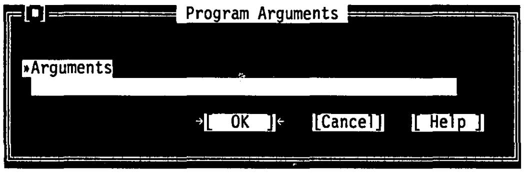
Arguments

The **Run | A**rguments command allows you to give your running programs command-line arguments exactly as if you had typed them on the DOS command line. DOS redirection commands will be ignored.

When you choose this command, a dialog box appears with a single input box.

Figure 2.9
The Arguments dialog box

You only need to enter the arguments here, not the program name.



Arguments take affect only when your program is started. If you are already debugging and wish to change the arguments, then you can select **P**rogram Reset to start the program with the new arguments.

Compile menu

Alt C

Use the commands on the **C**ompile menu to compile the program in the active window or to make or build your project. To use the **C**ompile, **M**ake, **B**uild, and **L**ink commands, you must have a file open in an *active* Edit window or a project defined (for **M**ake, **B**uild, and **L**ink). For example, if you open a Message or Watch window, those selections will be disabled.

Compile to OBJ

Alt F9

The **C**ompile | **C**ompile to OBJ command compiles the active editor file (a .C or .CPP file to an .OBJ file). The menu always displays the name of the file to be created; for example,

```
Compile to OBJ     C:EXAMPLE.OBJ
```

When Borland C++ is compiling, a status box pops up to display the compilation progress and results. When compiling/linking is complete, press any key to remove this box. If any errors or warnings occurred, the Message window becomes active and displays and highlights the first error.

Make EXE File

F9

The **C**ompile | **M**ake EXE File command invokes the Project Manager to make an .EXE file. The menu always displays the name of the .EXE file to be created; for example,

```
Make EXE File      C:EXAMPLE.EXE
```

The .EXE file name listed is derived from one of two names in the following order:

For more information on the Project Manager, see Chapter 4, "Managing multi-file projects."

- the project file (.PRJ) specified with the **P**roject | **O**pen Project command
- the name of the file in the active Edit window (if no project is defined, you'll get the default project defined by the file TCDEF.DPR)

Compile | **M**ake EXE File rebuilds only the files that aren't current.

Link EXE File

The **C**ompile | **L**ink EXE File command takes the current .OBJ and .LIB files (either the defaults or those defined in the current project file) and links them without doing a make; this produces a new .EXE file.

Build All

The **C**ompile | **B**uild All command rebuilds all the files in your project regardless of whether they're out of date.

This command is similar to **C**ompile | **M**ake EXE File except that it is unconditional. It performs the following steps:

1. It deletes the appropriate precompiled header (.SYM) file, if it exists.
2. It deletes any cached autodependency information in the project.
3. It sets the date and time of all the project's .OBJ files to zero
4. Finally, it does a make.

If you abort a **B**uild All command by pressing *Ctrl-Break* or get errors that stop the build, you can pick up where it left off simply by choosing **C**ompile | **M**ake EXE File.

Remove Messages

The **C**ompile | **R**emove Messages command removes all messages from the Message window.

Debug menu

Alt D

The commands on the **D**ebug menu control all the features of the integrated debugger. You can change default settings for these commands in the **O**ptions | De**b**ugger dialog box.

Windows

You can't run or debug Windows applications within the IDE. If you try to do so, you'll get an error dialog box to that effect. You must run them under Microsoft Windows and use Turbo Debugger for Windows.

Inspect

Alt F4

The **Debug | Inspect** command opens an Inspector window that lets you examine and modify values in a data element. The type of element you're inspecting determines the type of information presented in the window. In Borland C++, you can inspect simple (ordinal) data types like **char** or **unsigned long**, pointers, arrays, structures, classes, types, unions, and functions.

There are two ways to open an Inspector window:

- You can position the cursor on the data element you want to inspect, then choose *Alt-F4*.
- You can also choose **Debug | Inspect** to bring up the Inspector dialog box, and then type in the variable or expression you want to inspect. Alternatively, you can position the cursor on an expression, select **Debug | Inspect**, and, while in this dialog box, press → to bring in more of the expression. Press *Enter* to inspect it.

To close an Inspector window, make sure the window is active (topmost) and press *Esc* or choose **Window | Close**.

Here are some additional inspection operations you can perform:

- *Sub-inspecting*: Once you're in an Inspector window, you can inspect certain elements to isolate the view. When an inspector item is inspectable, the status line displays the message "↵ Inspect." To sub-inspect an item, you move the inspect bar to the desired item and press *Enter*.
- *Modifying inspector items*: When an inspector item can be modified, the status line displays "Alt-M Modify Field." Move the cursor to the desired item and press *Alt-M*; a dialog box will prompt you for the new value.
- *Range-inspect*: When you are inspecting certain elements, you can change the range of values that is displayed. For example, you can range-inspect pointer variables to tell Borland C++ how many elements the pointer points to. You can range-inspect an inspector when the status line displays the message "Set index range" and the command *Alt-I*.

The following sections briefly describe the eight types of Inspector windows possible.

Ordinal Inspector windows

Ordinal Inspector windows show you the value of simple data items, such as

```
char x = 4;
unsigned long y = 123456L;
```

These Inspector windows only have a single line of information following the top line (which usually displays the address of the variable, though it may display the word "constant" or have other information in it, depending on what you're inspecting). To the left appears the type of the scalar variable (**char, unsigned long,** and so forth), and to the right appears its present value. The value can be displayed as decimal, hex, or both. It's usually displayed first in decimal, with the hex values in parentheses (using the standard C hex prefix of 0x).

If the variable being displayed is of type **char**, the character equivalent is also displayed. If the present value does not have a printing character equivalent, the backslash (\) followed by a hex value displays the character value. This character value appears before the decimal or hex values.

Pointer Inspector windows

Pointer Inspector windows show you the value of data items that point to other data items, such as

```
char *p = "abc";
int *ip = 0;
int **ipp = &ip;
```

Pointer Inspector windows usually have a top line that contains the address of the pointer variable and the address being pointed to, followed by a single line of information.

To the left appears [0], indicating the first member of an array. To the right appears the value of the item being pointed to. If the value is a complex data item such as a structure or an array, as much of it as possible is displayed, with the values enclosed in braces ({ and }).

If the pointer is of type **char** and appears to be pointing to a null-terminated character string, more information appears, showing the value of each item in the character array. To the left in each line appears the array index ([1], [2], and so on), and the value appears to the right as it would in a scalar Inspector window. In this case, the entire string is also displayed on the top line, along with the address of the pointer variable and the address of the string that it points to.

Array Inspector windows

Array Inspector windows show you the value of arrays of data items, such as

```
long thread[3][4][5];
char message[] = "eat these words";
```

There is a line for each member of the array. To the left on each line appears the array index of the item. To the right appears the value of the item being pointed to. If the value is a complex data item such as a structure or array, as much of it as possible is displayed, with the values enclosed in braces ({ and }).

Structure and union Inspector windows

Structure and union Inspector windows show you the value of the members in your structure, class, and union data items. For example,

```
struct date {
   int year;
   char month;
   char day;
} today;

union {
   int small;
   long large;
} holder;
```

Structures and unions appear the same in Inspector windows. These Inspector windows have as many items after the address as there are members in the structure or union. Each item shows the name of the member on the left and its value on the right, displayed in a format appropriate to its C data type.

Function Inspector windows

Function Inspector windows show the return type of the function at the bottom of the inspector. Each parameter that a function is called with appears after the memory address at the top of the list.

Function Inspector windows give you information about the calling parameters, return data type, and calling conventions for a function.

Class Inspector windows

The Class (or object) Inspector window lets you inspect the details of a class variable. The window displays names and values for members and methods defined by the class.

The window can be divided into two panes horizontally, with the top pane listing the data fields or members of the class, and the bottom pane listing the member function names and the function addresses. Press *Tab* to move between the two panes of the Class Inspector window.

If the highlighted data field is a class or a pointer to a class, pressing *Enter* opens another Class Inspector window for the highlighted type. In this way, you can quickly inspect complex nested structures of classes with a minimum of keystrokes.

Constant Inspector window

Constant Inspector windows are much like Ordinal Inspector windows, but they have no address and can never be modified.

Type Inspector window

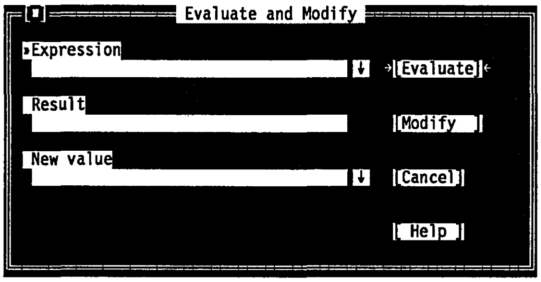
The Type Inspector window lets you examine a type. There is a Type Inspector window for each kind of instance inspector described here. The difference between them is that instance inspectors display the *value* of a field and type inspectors display the *type* of a field.

Evaluate/Modify

[Ctrl][F4]

The **Debug | Evaluate/Modify** command evaluates a variable or expression, displays its value, and, if appropriate, lets you modify the value. The command opens a dialog box containing three fields: the Expression field, the Result field, and the New Value field. Here is what the dialog box looks like:

Figure 2.10
The Evaluate/Modify dialog box



The Evaluate button is the default button; when you tab to the New Value field, the Modify button becomes the default.

The Expression field shows a default expression consisting of the word at the cursor in the Edit window. You can evaluate the default expression by pressing *Enter*, or you can edit or replace it first. You can also press → to extend the default expression by copying additional characters from the Edit window.

You can evaluate any valid C expression that doesn't contain

- function calls
- symbols or macros defined with **#define**
- local or static variables not in the scope of the function being executed

If the debugger can evaluate the expression, it displays the value in the Result field. If the expression refers to a variable or simple data element, you can move the cursor to the New Value field and enter an expression as the new value.

Press *Esc* to close the dialog box. If you've changed the contents of the New Value field but do not select Modify, the debugger will ignore the New Value field when you close the dialog box.

Use a repeat expression to display the values of consecutive data elements. For example, for an array of integers named *xarray*,

- `xarray[0],5` displays five consecutive integers in decimal.
- `xarray[0],5x` displays five consecutive integers in hexadecimal.

An expression used with a repeat count must represent a single data element. The debugger views the data element as the first element of an array if it isn't a pointer, or as a pointer to an array if it is.

The **D**ebug | **E**valuate/Modify command displays each type of value in an appropriate format. For example, it displays an **int** as an integer in base 10 (decimal), and an array as a pointer in base 16 (hexadecimal). To get a different display format, precede the expression with a comma followed by one of the format specifiers shown in Table 2.2.

Call Stack

Ctrl F3

The **D**ebug | **C**all Stack command opens a dialog box containing the call stack. The Call Stack window shows the sequence of functions your program called to reach the function now running.

At the bottom of the stack is **main**; at the top is the function that's now running.

Each entry on the stack displays the name of the function called and the values of the parameters passed to it.

*Compiling with Standard Stack Frame unchecked (**O | C | Code Generation**) causes some functions to be omitted from the call stack. Overlays can have the same effect. For more details, see page 66.*

Initially the entry at the top of the stack is highlighted. To display the current line of any other function on the call stack, select that function's name and press *Enter*. The cursor moves to the line containing the call to the function next above it on the stack.

For example, suppose the call stack looked like this:

```
func2()
func1()
main()
```

This tells you that **main** called **func1**, and **func1** called **func2.** If you wanted to see the currently executing line of **func1**, you could select **func1** in the call stack and press *Enter*. The code for **func1** would appear in the Edit window, with the cursor positioned on the call to **func2**.

To return to the current line of the function now being run (that is, to the run position), select the topmost function in the call stack and press *Enter*.

Table 2.2: Format specifiers recognized in debugger expressions

Character	Function
C	**Character.** Shows special display characters for control characters (ASCII 0 through 31); by default, such characters are shown using the appropriate C escape sequences (**\n**, **\t**, and so on). Affects characters and strings.
S	**String.** Shows control characters (ASCII 0 through 31) as ASCII values using the appropriate C escape sequences. Since this is the default character and string display format, the **S** specifier is only useful in conjunction with the **M** specifier.
D	**Decimal.** Shows all integer values in decimal. Affects simple integer expressions as well as arrays and structures containing integers.
H or X	**Hexadecimal.** Shows all integer values in hexadecimal with the 0*x* prefix. Affects simple integer expressions as well as arrays and structures containing integers.
F*n*	**Floating point.** Shows *n* significant digits (*n* is an integer between 2 and 18). The default value is 7. Affects only floating-point values.
M	**Memory dump.** Displays a memory dump, starting with the address of the indicated expression. The expression must be a construct that would be valid on the left side of an assignment statement, i.e., a construct that denotes a memory address; otherwise, the **M** specifier is ignored. By default, each byte of the variable is shown as two hex digits. Adding a **D** specifier with the **M** causes the bytes to be displayed in decimal. Adding an **H** or **X** specifier causes the bytes to be displayed in hex. An **S** or a **C** specifier causes the variable to be displayed as a string (with or without special characters). The default number of bytes displayed corresponds to the size of the variable, but a repeat count can be used to specify an exact number of bytes.
P	**Pointer.** Displays pointers in *seg*:*ofs* format with additional information about the address pointed to, rather than the default hardware-oriented *seg*:*ofs* format. Specifically, it tells you the region of memory in which the segment is located, and the name of the variable at the offset address, if appropriate. The memory regions are as follows:

Memory region	Evaluate message
0000:0000-0000:03FF	Interrupt vector table
0000:0400-0000:04FF	BIOS data area
0000:0500-Borland C++	MS-DOS/TSRs
Borland C++—User Program PSP	Borland C++
User Program PSP	User Process PSP
User Program—top of RAM	Name of a static user variable if its address falls inside the variable's allocated memory; otherwise nothing
A000:0000-AFFF:FFFF	EGA/VGA Video RAM
B000:0000-B7FF:FFFF	Monochrome Display RAM
B800:0000-BFFF:FFFF	Color Display RAM
C000:0000-EFFF:FFFF	EMS Pages/Adaptor BIOS ROMs
F000:0000-FFFF:FFFF	BIOS ROMs

Character	Function
R	**Structure/Union.** Displays field names as well as values, such as { *X:1, Y:10, Z:5* }. Affects only structures and unions.

Watches

The **D**ebug | **W**atches command opens a pop-up menu of commands that control the use of watch expressions. Watch expressions can be saved across sessions; see **O**ptions | **E**nvironment | **D**esktop. The following sections describe the commands in this pop-up menu.

Add Watch

Ctrl F7

The **A**dd Watch command inserts a watch expression into the Watch window.

When you choose this command, the debugger opens a dialog box and prompts you to enter a watch expression. The default expression is the word at the cursor in the current Edit window. There's also a history list available if you want to quickly enter an expression you've used before.

When you type a valid expression and press *Enter* or click OK, the debugger adds the expression and its current value to the Watch window. If the Watch window is the active window, you can insert a new watch expression by pressing *Ins*.

Delete Watch

The **D**elete Watch command deletes the current watch expression from the Watch window (which must be the active window in order to use this command). To delete a watch expression other than the current one, select the desired watch expression and press *Del* or *Ctrl-Y*.

Edit Watch

The **E**dit Watch command allows you to edit the current watch expression in the Watch window. A history list is available to save you time retyping.

When you choose this command, the debugger opens a dialog box containing a copy of the current watch expression. Edit the expression and press *Enter*. The debugger replaces the original version of the expression with the edited one.

You can also edit a watch expression from inside the Watch window by selecting the expression and pressing *Enter*.

Remove All Watches

The **R**emove All Watches command deletes all watch expressions from the Watch window.

Toggle Breakpoint

Ctrl F8

The **D**ebug | **T**oggle Breakpoint command lets you set or clear an unconditional breakpoint on the line where the cursor is positioned. When a breakpoint is set, it is marked by a breakpoint highlight. Breakpoints can be saved across sessions using **O**ptions | **E**nvironment | **D**esktop. See the following section for more information on breakpoints.

Breakpoints

The **D**ebug | **B**reakpoints command opens a dialog box that lets you control the use of breakpoints—both conditional and unconditional ones. Here is what the dialog box looks like:

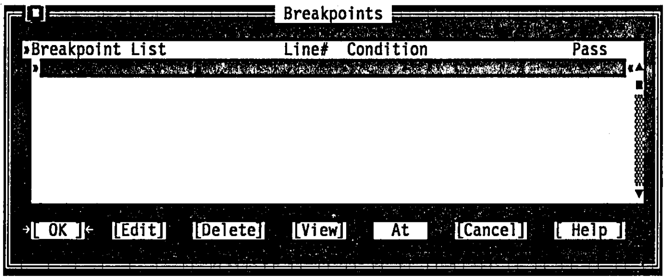
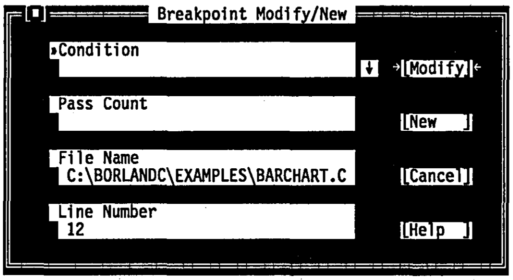


Figure 2.11
The Breakpoints dialog box

The dialog box shows you all set breakpoints, their line numbers, and the conditions. The condition has a history list so you can select a breakpoint condition that you've used before.

You can remove breakpoints from your program by choosing the Delete button. You can also view the source where existing breakpoints are set by choosing the View button. View moves the cursor to the selected breakpoint. This command does not run your code; it only positions the cursor at active breakpoints in the Edit window.

Choose Edit to add the new one to the list. When you edit a breakpoint, this dialog box appears over the first one:



Condition

Pass Count

File Name
C:\BORLANDC\EXAMPLES\BARCHART.C

Line Number
12

Figure 2.12
The Breakpoint Modify/New dialog box

Again, line number and conditions are that of the breakpoints you've set. Use Pass Count to set how many times the breakpoint should be skipped before stopping. The At button lets you specify a breakpoint at a particular function (you must be debugging to access this).

This dialog box also has a New button, which lets you enter breakpoint information for a new breakpoint, and a Modify button, which accepts the settings of the box.

Your program stops wherever it encounters a breakpoint in the course of running. When the program stops, the run bar is on the line containing the breakpoint. (The breakpoint highlight is obscured by the run bar; it reappears when the run bar moves on.)

When a source file is edited, each breakpoint "sticks" to the line where it is set. Breakpoints stay set until you:

- delete the source line a breakpoint is set on
- clear a breakpoint with **T**oggle Breakpoint

Borland C++ will attempt to track breakpoints in two cases:

- If you edit a file containing breakpoints and then don't save the edited version of the file.
- If you edit a file containing breakpoints and then continue the current debugging session without remaking the program. (Borland C++ displays the warning prompt "Source modified, rebuild?")

Before you compile a source file, you can set a breakpoint on any line, even a blank line or a comment. When you compile and run the file, Borland C++ validates any breakpoints that are set and gives you a chance to remove, ignore, or change invalid breakpoints. When you are debugging the file, Borland C++ knows which lines contain executable statements, and will warn you if you try to set invalid breakpoints.

You can set an unconditional breakpoint without going through the dialog box by choosing the **D**ebug | **T**oggle Breakpoint command.

Project menu

[Alt][P]

The **P**roject menu contains all the project management commands to

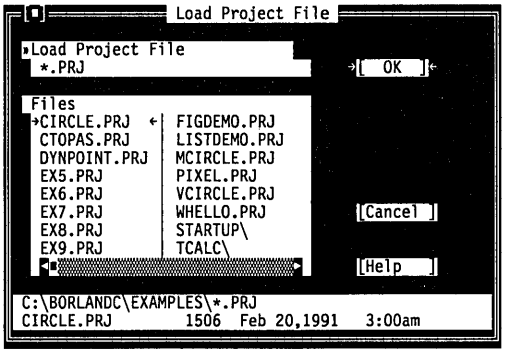
If you have project files in Turbo C 2.0, you can convert them to Borland C++ format using the standalone utility PRJCNVT. See UTIL.DOC for details.

- create a project
- add or delete files from your project (for examples on how to use the Project Manager, see Chapter 4, "Managing multi-file projects")
- specify which program your source file should be translated with
- set options for a file in the project
- specify which command-line override options to use for the translator program
- specify what the resulting object module is to be called, where it should be placed, whether the module is an overlay, and whether the module should contain debug information
- view included files for a specific file in the project

Open Project

The **O**pen Project command displays the Load Project File dialog box, which allows you to select and load a project or create a new project by typing in a name.

Figure 2.13
The Project File dialog box



```
*.PRJ

CIRCLE.PRJ        FIGDEMO.PRJ
CTOPAS.PRJ        LISTDEMO.PRJ
DYNPOINT.PRJ      MCIRCLE.PRJ
EX5.PRJ           PIXEL.PRJ
EX6.PRJ           VCIRCLE.PRJ
EX7.PRJ           WHELLO.PRJ
EX8.PRJ           STARTUP\
EX9.PRJ           TCALC\

C:\BORLANDC\EXAMPLES\*.PRJ
CIRCLE.PRJ        1506  Feb 20,1991    3:00am
```

This dialog box lets you select a file name similar to the **File | Open** dialog box, discussed on page 28. The file you select will be used as a project file, which is a file that contains all the information needed to build your project's executable. Borland C++ uses the project name when it creates the .EXE and .MAP files. A typical project file has the extension .PRJ.

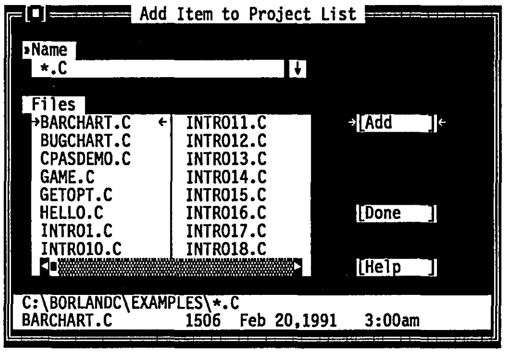
Close Project

Choose **Project | Close** Project when you want to remove your project and return to the default project (TCDEF.DPR).

Add Item

Choose **Project | Add** Item when you want to add a file to the project list. This brings up the Add Item to Project List dialog box:

Figure 2.14
The Add Item to Project List dialog box



```
BARCHART.C      INTRO11.C
BUGCHART.C      INTRO12.C
CPASDEMO.C      INTRO13.C
GAME.C          INTRO14.C
GETOPT.C        INTRO15.C
HELLO.C         INTRO16.C
INTRO1.C        INTRO17.C
INTRO10.C       INTRO18.C

C:\BORLANDC\EXAMPLES\*.C
BARCHART.C        1506   Feb 20,1991    3:00am
```

This dialog box is set up much like the Load File dialog box (**File | Open**). Choosing the Add button puts the currently highlighted file in the Files list into the Project window. The chosen file is added to the Project window File list immediately after the highlight bar in the Project window. The highlight bar is advanced each time a file is added. (When the Project Window is active, you can press *Ins* to add a file.)

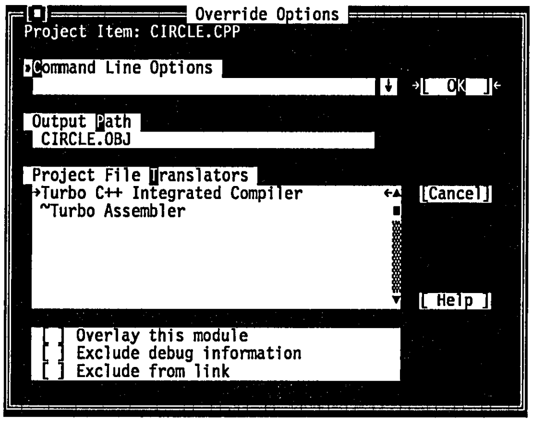
Delete Item

Choose **Project | Delete** Item when you want to delete a file in the Project window. When the Project window is active, you can press *Del* to delete a file.

Local Options

The Local Options command opens the following dialog box:

Figure 2.15 The Override Options dialog box


```
Override Options
Project Item: CIRCLE.CPP
Command Line Options

Output Path
 CIRCLE.OBJ
Project File Translators
→Turbo C++ Integrated Compiler
 ~Turbo Assembler

[ ] Overlay this module
[ ] Exclude debug information
[ ] Exclude from link
```

These command-line options are not supported: ***c***, ***E****filename*, ***e***, ***I****pathname*, ***L****x*, ***M***, ***Q***, ***y***.

The Override Options dialog box lets you include command-line override options for a particular project-file module. It also lets you give a specific path and name for the object file and lets you choose a translator for the module.

Any program you installed in the Transfer dialog box with the Translator option checked appears in the list of Project File Translators (see page 74 for information on the Transfer dialog box).

`[ ] Overlay this module`

Check the Overlay this Module option if you want the selected module or library (or project item) to be overlaid. This item is local to one file. It is disabled if the Overlay support checkbox is not marked (in **O**ptions | **C**ompile | **C**ode Generation).

`[ ] Exclude debug information`

Check the Exclude Debug Information option to prevent debug information included in the module you've selected from going into the .EXE.

Use this switch on already debugged modules of large programs. You can change which modules have debug information simply by checking this box and then re-linking (no compiling is required).

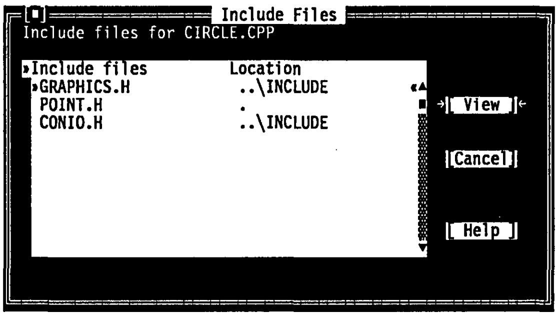
`[ ] Exclude from link`

Check the Exclude from Link option if you don't want this module linked in.

Include Files

Choose **P**roject | **I**nclude Files to display the Include Files dialog box; do this when you want to see which files are included by the file you chose from the Project window. When you're in the Project Window, you can press *Spacebar* to display the Include Files dialog box. This command is disabled if you've yet to build a project.

The Include Files dialog box looks like this:



```
Include Files
Include files for CIRCLE.CPP

Include files        Location
GRAPHICS.H           ..\INCLUDE
POINT.H              .
CONIO.H              ..\INCLUDE
```

Figure 2.16
The Include Files dialog box

After a file has been compiled, information is collected about that file (notice that the Project window has code size information). In this state, the Project manager also knows which include file the module references. You can view the active Edit window's include files in the Include Files dialog box. From the Project Manager window, press *Spacebar* to display the dialog box. From an Edit window, go to the **P**roject menu and choose **I**nclude Files. You can scroll through the list of files displayed. The default action is to view the selected file, so pressing *Enter* opens that include file into an Edit window.

Options menu

[Alt][O]

The **O**ptions menu contains commands that let you view and change various default settings in Borland C++. Most of the commands in this menu lead to a dialog box.

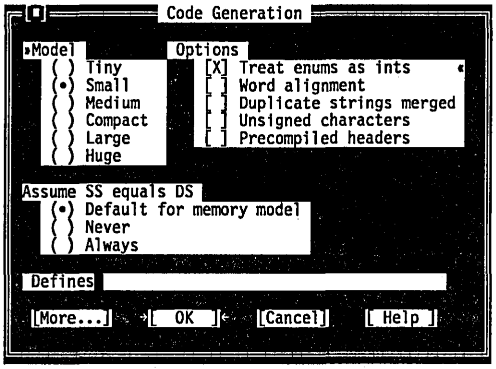
Compiler

The **O**ptions I **C**ompiler command displays a pop-up menu that gives you several options to set that affect code compilation. The following sections describe these commands.

Code Generation

The **C**ode Generation command displays a dialog box. The settings in this box tell the compiler to prepare the object code in certain ways. The dialog box looks like this:

Figure 2.17
The Code Generation dialog box



```
Code Generation
Model              Options
 ( ) Tiny           [X] Treat enums as ints
 (•) Small          [ ] Word alignment
 ( ) Medium         [ ] Duplicate strings merged
 ( ) Compact        [ ] Unsigned characters
 ( ) Large          [ ] Precompiled headers
 ( ) Huge

Assume SS equals DS
 (•) Default for memory model
 ( ) Never
 ( ) Always

Defines

[More...]   [ OK ]   [Cancel]   [ Help ]
```

Here are what the various buttons and check boxes mean:

```
Model
 ( ) Tiny
 (•) Small
 ( ) Medium
 ( ) Compact
 ( ) Large
 ( ) Huge
```

The Model buttons determine which memory model you want to use. The memory model chosen determines the default method of memory addressing. The default memory model is Small. Refer to Chapter 6, "Memory management," in the *Programmer's Guide* for more information about memory models in general. There are some restrictions about which memory models you can use effectively for different types of Windows executables; see page 115 in Chapter 3 for a discussion of those restrictions.

The options control various code generation defaults.

```
Options
[X] Treat enums as ints
[ ] Word alignment
[ ] Duplicate strings merged
[ ] Unsigned characters
[ ] Precompiled headers
```

- When checked, Treat enums as ints causes the compiler to always allocate a whole word. Unchecked, this option tells the compiler to allocate an unsigned or signed byte if the minimum and maximum values of the enumeration are both within the range of 0 to 255 or –128 to 127, respectively.
- Word Alignment (when checked) tells Borland C++ to align noncharacter data (within structures and unions only) at even addresses. When this option is off (unchecked), Borland C++

uses byte-aligning, where data (again, within structures and unions only) can be aligned at either odd or even addresses, depending on which is the next available address.

Word alignment increases the speed with which 80x86 processors fetch and store the data.

- Duplicate Strings Merged (when checked) tells Borland C++ to merge two strings when one matches another. This produces smaller programs, but can introduce bugs if you modify one string.
- Unsigned Characters (when checked) tells Borland C++ to treat all **char** declarations as if they were **unsigned char** type.

See Appendix A for more on precompiled headers.

- Check Precompiled Headers when you want the IDE to generate and use precompiled headers. Precompiled headers can dramatically increase compilation speeds, though they require a considerable amount of disk space. When this option is off (the default), the IDE will neither generate nor use precompiled headers. Precompiled headers are saved in *PROJECTNAME*.SYM.

```
Assume SS equals DS
 (•) Default for memory model
 ( ) Never
 ( ) Always
```

If the Default For Memory Model radio button is pushed, whether the stack segment (SS) is assumed to be equal to the data segment (DS) is dependent on the memory model used. Usually, the compiler assumes that SS is equal to DS in the small, tiny, and medium memory models (except for DLLs). See pages 112 and 166 for detailed and specific explanations of what the compiler assumes for each memory model and for Windows applications and DLLs.

When the Never radio button is pushed, the compiler will not assume SS is equal to DS.

The Always button tells the compiler to always assume that SS is equal to DS. It causes the IDE to substitute the C0Fx.OBJ startup module for C0x.OBJ to place the stack in the data segment.

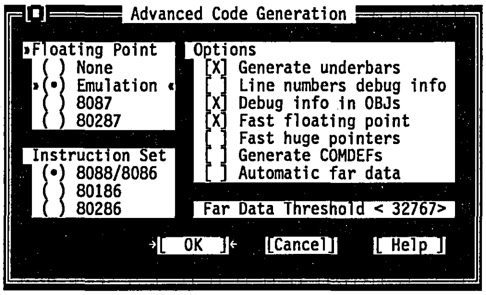
```
Defines
```

Use the Defines input box to enter macro definitions to the preprocessor. You can separate multiple defines with semicolons (;) and assign values with an equal sign (=); for example,

```
TESTCODE;PROGCONST=5
```

Leading and trailing spaces will be stripped, but embedded spaces are left intact. If you want to include a semicolon in a macro, you must place a backslash (\) in front of it.

The Code Generation dialog box has a button called More that takes you to the Advanced Code Generation dialog box. Here's what that dialog box looks like:



```
Advanced Code Generation

Floating Point          Options
  ( ) None                [X] Generate underbars
  (•) Emulation           [ ] Line numbers debug info
  ( ) 8087                [X] Debug info in OBJs
  ( ) 80287               [X] Fast floating point
                          [ ] Fast huge pointers
Instruction Set           [ ] Generate COMDEFs
  (•) 8088/8086           [ ] Automatic far data
  ( ) 80186
  ( ) 80286               Far Data Threshold < 32767>

        [ OK ]     [Cancel]     [ Help ]
```

Figure 2.18
The Advanced Code Generation dialog box

```
Floating Point
  ( ) None
  (•) Emulation
  ( ) 8087
  ( ) 80287
```

The Floating Point buttons let you decide how you want Borland C++ to handle floating-point numbers.

- Choose None if you're not using floating point. (If you choose None and you use floating-point calculations in your program, you get link errors.)
- Choose Emulation if you want your program to detect whether your computer has an 80x87 coprocessor (and to use it if you do). If it is not present, your program will emulate the 80x87.
- Choose 8087 or 80287 to generate direct 8087 or 80287 inline code.

```
Instruction Set
  (•) 8088/8086
  ( ) 80186
  ( ) 80286
```

The Instruction Set radio buttons let you choose what instruction set to generate code for. The default instruction set, 8088/8086, works with all PCs.

The advanced options are shown to the left.

```
Options
  [X] Generate underbars
  [ ] Line numbers debug info
  [X] Debug info in OBJs
  [X] Fast floating point
  [ ] Fast huge pointers
  [ ] Generate COMDEFs
  [ ] Automatic far data
```

- When checked, the Generate Underbars option automatically adds an underbar, or underscore, character (_) in front of every global identifier (that is, functions and global variables). If you are linking with standard libraries, this box must be checked.
- Line Numbers Debug Info (when checked) includes line numbers in the object and object map files (the latter for use by a symbolic debugger). This increases the size of the object and map files but does not affect the speed of the executable program.

  Since the compiler might group together common code from multiple lines of source text during jump optimization, or might reorder lines (which makes line-number tracking difficult), you might want to make sure the Jump Optimization check box (**O**ptions | **C**ompiler | **O**ptimizations) is off (unchecked) when this option is checked.

- Debug Info in OBJs controls whether debugging information is included in object (.OBJ) files. The default for this check box is on (checked), which you need in order to use both the integrated debugger and the standalone Turbo Debugger.
- This option allows you to link and create larger object files. While this option doesn't affect execution speed, it *does* affect compilation time. We recommend that you have this option checked. If you want debugging information with this option checked, turn off debug information in **O**ptions | **D**ebugging, or, even better, link with debugging information, then use `TDSTRIP -s` to strip it off to a separate .TDS file for debugging. TDSTRIP is documented in the online file called UTIL.DOC.
- Fast Floating Point lets you optimize floating-point operations without regard to explicit or implicit type conversions. When this option is unchecked, the compiler follows strict ANSI rules regarding floating-point conversions.

See page 169 for more details on fast huge pointers.

- The Fast Huge Pointers option normalizes huge pointers only when a segment wrap-around occurs in the offset portion of the segment. This greatly speeds up the computation of huge pointer expressions, but must be used with caution, as it can cause problems for huge arrays if array elements cross a segment boundary.
- When checked, the Generate COMDEFs option allows a definition of a variable to appear in header files as long as it is not initialized. Thus a definition such as `int SomeArray[256];` could appear in a header file that is then included in many modules, and the compiler will generate it as a communal variable (a COMDEF record rather than a PUBDEF record). The linker will then only generate one instance of the variable so it will not be a duplicate definition linker error.

This option is ignored if you're using the tiny, small, or medium memory models.

- The Automatic Far Data option and the Far Data Threshold type-in box work together. When checked, the Automatic Far Data option tells the compiler to automatically generate far objects; the Far Data Threshold < 32767> specifies the size portion needed to complete the command. The size value is ignored if Automatic Far Data is not checked.

Entry/Exit Code

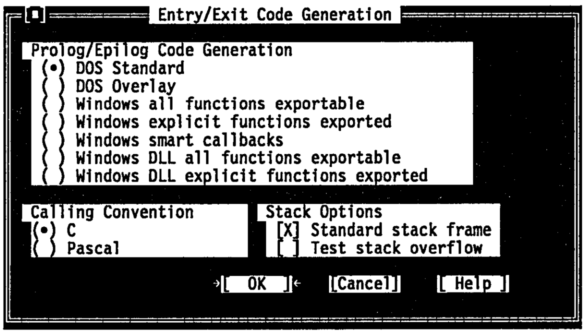
See Chapter 3 for more on prolog and epilog code.

When you compile a C or C++ program for Windows or DOS, the compiler needs to know which kind of prolog and epilog to create for each of a module's functions.

If the program is intended for Windows, the compiler generates a different prolog and epilog than it would for DOS. Because of this, you must use the the Entry/Exit Code Generation dialog box to set the appropriate application. If you use the Set Application Options dialog box (described on page 86), the settings in the Entry/Exit Code dialog box will already be correct for the type of application you choose.

This dialog box also allows you to select the calling convention and to set a couple of stack options. All options affect what code is generated for function calls and returns.

Figure 2.19
The Entry/Exit Code dialog box


```
Entry/Exit Code Generation

Prolog/Epilog Code Generation
 (•) DOS Standard
 ( ) DOS Overlay
 ( ) Windows all functions exportable
 ( ) Windows explicit functions exported
 ( ) Windows smart callbacks
 ( ) Windows DLL all functions exportable
 ( ) Windows DLL explicit functions exported

Calling Convention            Stack Options
(•) C                          [X] Standard stack frame
( ) Pascal                     [ ] Test stack overflow

                 [ OK ]     [Cancel]     [ Help ]
```

You can set prolog/epilog code for DOS *or* for Windows, but not for both.

If you want to set the prolog/epilog code for a DOS application, you need to select DOS Standard or DOS Overlay.

- Push the DOS Standard radio button to tell the compiler to generate code that may not be safe for overlays. If you don't plan to create an overlaid application, use this option.
- Push the DOS Overlay radio button to tell the compiler to generate overlay safe code. Use this option when you're creating an overlaid application.

If you want to set the prolog/epilog code for a Windows application, you need to select one of five options.

- Windows All Functions Exportable is the most general kind of Windows executable, although not necessarily the most efficient. It assumes that all functions are capable of being called by the Windows kernel or by other modules, and generates the necessary overhead information for every function, whether the function needs it or not. The module definition file will control which functions actually get exported.

- Use Windows Explicit Functions Exported if you have functions that will not be called the Windows kernel; it isn't necessary to generate export-compatible prolog/epilog code information for these functions. The **_export** keyword provides a way to tell the compiler which specific functions will be exported: Only those far functions with **_export** will be given the special Windows prolog/epilog code.
- Push the Windows Smart Callbacks button to select Borland C++ smart callbacks. See page 112 in Chapter 3 for details on smart callbacks.

  The compiler must be able to assume that DS = SS for all functions in the module. You therefore should *not* choose Smart Callbacks for a module that will be compiled under the huge memory model.
- Push the Windows DLL All Functions Exportable button to create an .OBJ file to be linked as a .DLL with all functions exportable.
- Push the Windows DLL Explicit Functions Exported button to create an .OBJ file to be linked as a .DLL with certain functions explicitly selected to be exported. Otherwise this is essentially the same as Windows Explicit Functions Exported, see that discussion for more.

```
Calling Convention
 (•) C
 ( ) Pascal
```

The Calling Convention options cause the compiler to generate either a C calling sequence or a Pascal calling sequence for function calls. The differences between C and Pascal calling conventions are in the way each handles stack cleanup, order of parameters, case, and prefix (underbar) of global identifiers.

Important! *Do not change this option unless you're an expert and have read Chapter 9, "Interfacing with assembly language," in the Programmer's Guide.*

```
Stack Options
 [X] Standard Stack Frame
 [ ] Test Stack Overflow
```

- Standard Stack Frame (when checked) generates a standard stack frame (standard function entry and exit code). This is helpful when debugging—it simplifies the process of tracing back through the stack of called subroutines.

  If you compile a source file with this option off (unchecked), any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code.

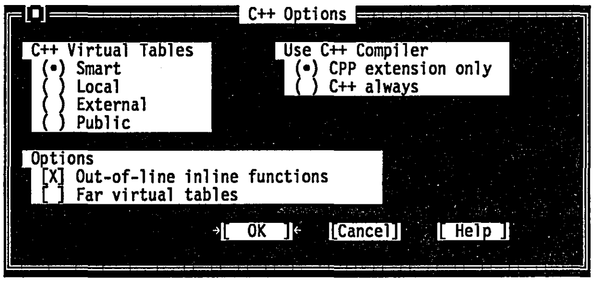
This makes the code shorter and faster, but prevents the **D**ebug | **C**all Stack command from "seeing" the function. Thus, you should always check the option when you compile a source file for debugging.

- When checked, the Test Stack Overflow generates code to check for a stack overflow at run time. Even though this costs space and time in a program, it can be a real lifesaver, since a stack overflow bug can be difficult to track down.

C++ Options

The C++ Options command displays a dialog box that contains settings that tell the compiler to prepare the object code in certain ways when using C++.

Figure 2.20
The C++ options dialog box



```
C++ Virtual Tables
 (•) Smart
 ( ) Local
 ( ) External
 ( ) Public

Use C++ Compiler
 (•) CPP extension only
 ( ) C++ always

Options
 [X] Out-of-line inline functions
 [ ] Far virtual tables
```

```
C++ Virtual Tables
 (•) Smart
 ( ) Local
 ( ) External
 ( ) Public
```

The C++ Virtual Tables radio buttons let you control C++ virtual tables and the expansion of inline functions when debugging.

- The Smart option generates C++ virtual tables (and inline functions not expanded inline) so that only one instance of a given virtual table or inline function will be included in the program. This produces the smallest and most efficient executables, but uses .OBJ (and .ASM) extensions only available with TLINK 3.0 and TASM 2.0 (or newer).
- The Local option generates local virtual tables (and inline functions not expanded inline) such that each module gets its own private copy of each virtual table or inline function it uses; this option uses only standard .OBJ (and .ASM) constructs, but produces larger executables.
- The External option generates external references to virtual tables; one or more of the modules comprising the program must be compiled with the Public option to supply the definitions for the virtual tables.
- The Public option generates public definitions for virtual tables.

```
Use C++ Compiler
 (•) CPP extension only
 ( ) C++ always
```

The Use C++ Compiler radio buttons tell Borland C++ whether to always compile your programs as C++ code, or to always compile your code as C code except when the file extension is .CPP.

```
[X] Out-of-line inline functions
[ ] Far virtual tables
```

- Use Out-of-Line Inline Functions when you want to step through or set breakpoints on inline functions.
- The Far Virtual Tables option causes virtual tables to be created in the code segment instead of the data segment, and makes virtual table pointers into full 32-bit pointers (the latter is done automatically if you are using the huge memory model).

  There are two primary reasons for using this option: to remove the virtual tables from the data segment, which may be getting full, and to be able to share objects (of classes with virtual functions) between modules that use different data segments (for example, a DLL and an executable using that DLL). You must compile all modules that may share objects either entirely with or entirely without this option. You can achieve the same effect by using the **huge** or **_export** modifiers on a class-by-class basis.

Optimizations

The **O**ptimizations command displays a dialog box. The settings in this box tell the compiler to prepare the object code in certain ways to optimize the size or speed. The dialog box looks like this:

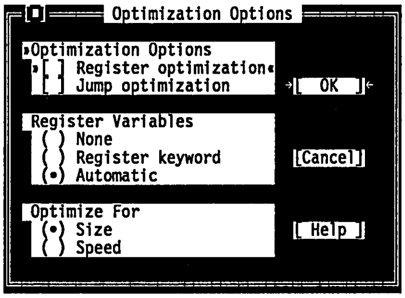


Figure 2.21 The Optimizations Options dialog box

```
Optimization Options
 [ ] Register optimization
 [ ] Jump optimization

Register Variables
 ( ) None
 ( ) Register keyword
 (•) Automatic

Optimize For
 (•) Size
 ( ) Speed
```

```
Optimization Options
 [ ] Register optimization
 [ ] Jump optimization
```

The Optimizations Options affect how optimization of your code occurs.

- Register Optimization 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 can't detect whether a value has been modified indirectly by a pointer.

- Jump Optimization reduces the code size by eliminating redundant jumps and reorganizing loops and switch statements.

Important!

When this option is checked, the sequences of tracing and stepping in the debugger can be confusing, since there might be multiple lines of source code associated with a particular generated code sequence. For best stepping results, turn this option off (uncheck it) while you are debugging.

```
Register Variables
 ( ) None
 ( ) Register keyword
 (•) Automatic
```

The Register Variables radio buttons suppress or enable the use of register variables.

With Automatic chosen, register variables are automatically assigned for you. With None chosen, the compiler does not use register variables even if you've used the **register** keyword. With Register keyword chosen, the compiler uses register variables only if you use the **register** keyword and a register is available. (See Chapter 6, "Memory management," in the *Programmer's Guide* for more details.)

Generally, you can keep this option set to Automatic unless you're interfacing with preexisting assembly code that does not support register variables.

```
Optimize For
 (•) Size
 ( ) Speed
```

The Optimize For buttons let you change Borland C++'s code generation strategy. Normally the compiler optimizes for size, choosing the smallest code sequence possible. You can also have the compiler optimize for speed, so that it chooses the *fastest* sequence for a given task. If you are creating Windows applications, normally you'll want to optimize for speed.

Source

The **S**ource command displays a dialog box. The settings in this box tell the compiler to expect certain types of source code. The dialog box looks like this:

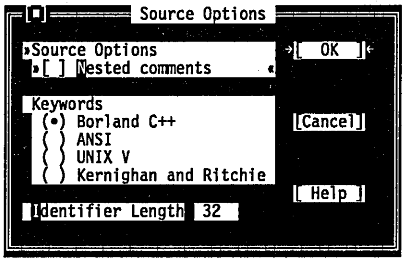

```
Source Options
Source Options
 [ ] Nested comments
Keywords
 (•) Borland C++
 ( ) ANSI
 ( ) UNIX V
 ( ) Kernighan and Ritchie
Identifier Length  32
```

Figure 2.22
The Source Options dialog box

```
Source Options
 [ ] Nested comments
```

The Nested Comments checkbox allows you to nest comments in Borland C++ source files. Nested comments are not allowed in standard C implementations, and they are not portable.

```
Keywords
 (•) Borland C++
 ( ) ANSI
 ( ) UNIX V
 ( ) Kernighan and Ritchie
```

The Keyword radio buttons tell the compiler how to recognize keywords in your programs.

- Choosing Borland C++ tells the compiler to recognize the Borland C++ extension keywords, including **near, far, huge, asm, cdecl, pascal, interrupt, _es, _export, _ds, _cs, _ss,** and the register pseudovariables (_AX, _BX, and so on). For a complete list, refer to Chapter 1, "Lexical grammar," in the *Programmer's Guide.*
- Choosing ANSI tells the compiler to recognize only ANSI keywords and treat any Borland C++ extension keywords as normal identifiers.
- Choosing UNIX V tells the compiler to recognize only UNIX V keywords and treat any Borland C++ extension keywords as normal identifiers.
- Choosing Kernighan and Ritchie tells the compiler to recognize only the K&R extension keywords and treat any Borland C++ extension keywords as normal identifiers.

```
Identifier Length  32
```

Use the Identifier Length input box to specify the number (*n*) of significant characters in an identifier. Except in C++, which recognizes identifiers of unlimited length, all identifiers are treated as distinct only if their first *n* characters are distinct. This includes variables, preprocessor macro names, and structure member names. The number can be from 1 to 32; the default is 32.

Messages

The **M**essages command displays a dialog box that lets you set several options that affect compiler error messages in the IDE.

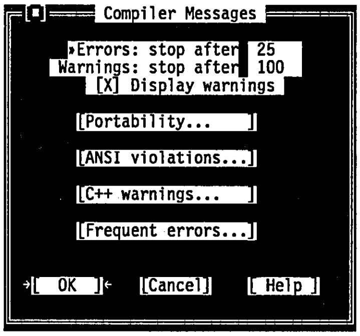


Figure 2.23 The Compiler Messages dialog box

```
Compiler Messages
 Errors: stop after    25
 Warnings: stop after  100
 [X] Display warnings
```

```
  Errors: stop after  25
Warnings: stop after  100

    [X] Display warnings
```

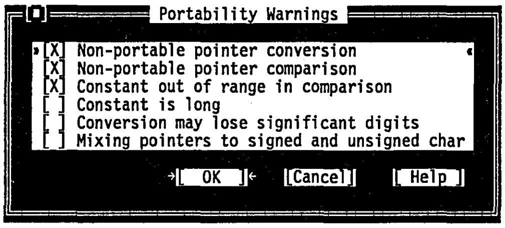
- The Errors: Stop After option causes compilation to stop after a specified number of errors have been detected. The default is 25, but you can enter any number from 0 to 255. (Entering 0 causes compilation to continue until the end of the file.)
- The Warnings: Stop After option causes compilation to stop after a specified number of warnings have been detected. The default is 100, but you can enter any number from 0 to 255. (Entering 0 causes compilation to continue until the end of the file or until the error limit has been reached, whichever comes first.)
- The Display Warnings option (when checked) means that any or all of the following warning types can be displayed if chosen:
  - Portability warnings
  - ANSI violations
  - C++ warnings
  - Frequent errors

When this option is off (unchecked), none of these warnings will be displayed.

```
Portability...
```

When you choose the Portability button in the Compiler Messages dialog box, another dialog box appears that lets you make specific settings in this category:

Figure 2.24
The Portability warnings dialog box



```
Portability Warnings

[X] Non-portable pointer conversion
[X] Non-portable pointer comparison
[X] Constant out of range in comparison
[ ] Constant is long
[ ] Conversion may lose significant digits
[ ] Mixing pointers to signed and unsigned char

    [ OK ]   [Cancel]   [ Help ]
```

Check the warnings you want to be notified of and uncheck the ones you don't. Choose OK to return to the Compiler Messages dialog box.

```
ANSI violations...
```

When you choose the ANSI Violations button in the Compiler Messages dialog box, another dialog box appears that lets you make specific settings in this category. Here is what this dialog box looks like:

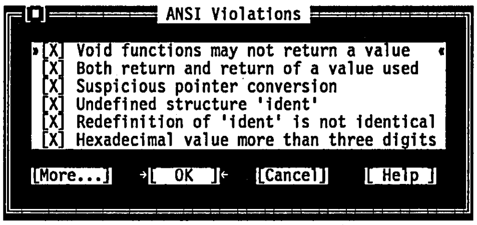


Figure 2.25
The ANSI Violations dialog box

```
ANSI Violations
[X] Void functions may not return a value
[X] Both return and return of a value used
[X] Suspicious pointer conversion
[X] Undefined structure 'ident'
[X] Redefinition of 'ident' is not identical
[X] Hexadecimal value more than three digits
```

Check the warnings you want to be notified of and uncheck the ones you don't. Choose OK to return to the Compiler Messages dialog box.

When you choose the More ANSI Violations button in the ANSI Violations dialog box, another dialog box appears with more settings you can make in this category:

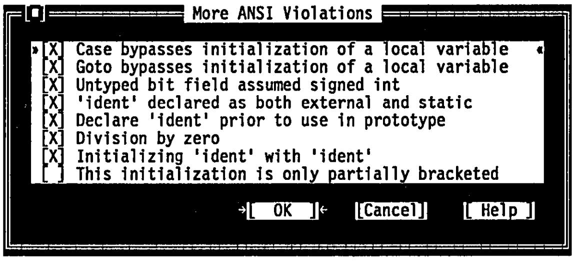


Figure 2.26
The More ANSI Violations dialog box

```
More ANSI Violations
[X] Case bypasses initialization of a local variable
[X] Goto bypasses initialization of a local variable
[X] Untyped bit field assumed signed int
[X] 'ident' declared as both external and static
[X] Declare 'ident' prior to use in prototype
[X] Division by zero
[X] Initializing 'ident' with 'ident'
[ ] This initialization is only partially bracketed
```

Check or uncheck these warnings just like in the previous dialog box's and choose OK to return to the ANSI Violations dialog box.

`C++ warnings...`

When you choose the C++ Warnings button in the Compiler Messages dialog box, another dialog box appears that lets you make specific settings in this category. Here is what this dialog box looks like:

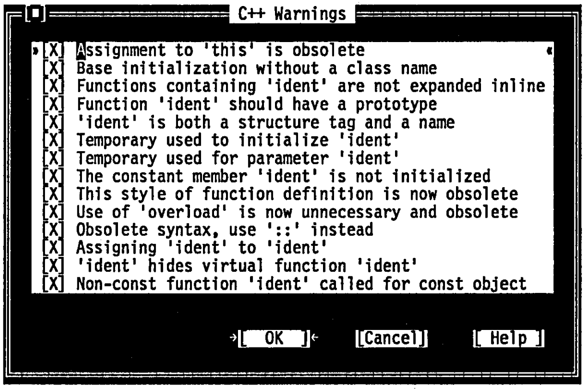


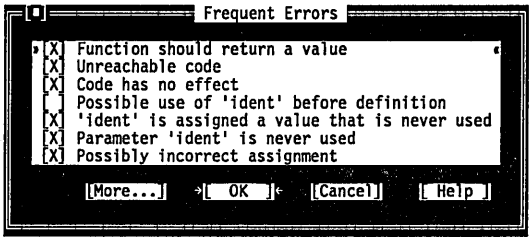
Figure 2.27
The C++ Warnings dialog box

```
C++ Warnings
[X] Assignment to 'this' is obsolete
[X] Base initialization without a class name
[X] Functions containing 'ident' are not expanded inline
[X] Function 'ident' should have a prototype
[X] 'ident' is both a structure tag and a name
[X] Temporary used to initialize 'ident'
[X] Temporary used for parameter 'ident'
[X] The constant member 'ident' is not initialized
[X] This style of function definition is now obsolete
[X] Use of 'overload' is now unnecessary and obsolete
[X] Obsolete syntax, use '::' instead
[X] Assigning 'ident' to 'ident'
[X] 'ident' hides virtual function 'ident'
[X] Non-const function 'ident' called for const object
```

Check the warnings you want to be notified of and uncheck the ones you don't. Choose OK to return to the Compiler Messages dialog box.

`Frequent errors...`

When you choose the Frequent Errors button in the Compiler Messages dialog box, another dialog box appears that lets you make specific settings in this category. Here is what this dialog box looks like:

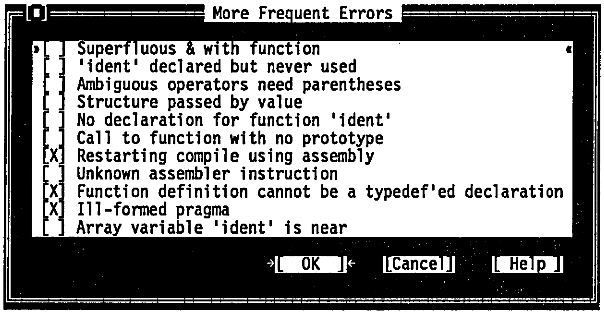


```
Frequent Errors

[X] Function should return a value
[X] Unreachable code
[X] Code has no effect
[ ] Possible use of 'ident' before definition
[X] 'ident' is assigned a value that is never used
[X] Parameter 'ident' is never used
[X] Possibly incorrect assignment

[More...]  [ OK ]  [Cancel]  [ Help ]
```

Figure 2.28
The Frequent Errors dialog box

Check the errors you want to be notified of and uncheck the ones you don't. Choose OK to return to the Compiler Messages dialog box.

Choosing the More button takes you to the More Frequent Errors dialog box:



```
More Frequent Errors

[ ] Superfluous & with function
[ ] 'ident' declared but never used
[ ] Ambiguous operators need parentheses
[ ] Structure passed by value
[ ] No declaration for function 'ident'
[ ] Call to function with no prototype
[X] Restarting compile using assembly
[ ] Unknown assembler instruction
[X] Function definition cannot be a typedef'ed declaration
[X] Ill-formed pragma
[ ] Array variable 'ident' is near

[ OK ]  [Cancel]  [ Help ]
```

Figure 2.29
The More Frequent Errors dialog box

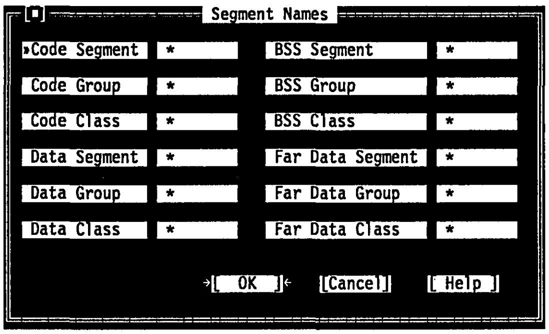
Check or uncheck these errors like in the previous dialog boxes and choose OK to return to the Frequent Errors dialog box.

Names

The **N**ames command brings up the following dialog box, which lets you change the default segment, group, and class names for code, data, and BSS sections. *Don't change the settings in this command unless you are an expert and have read Chapter 6, "Memory management," in the Programmer's Guide.*

Figure 2.30
The Segment Names dialog box

The asterisk means that the default names will be used if no other names are specified. See page 177 for the default names.

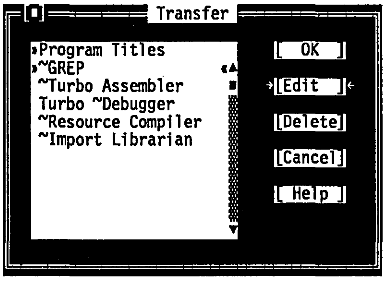


Code Segment	*	BSS Segment	*
Code Group	*	BSS Group	*
Code Class	*	BSS Class	*
Data Segment	*	Far Data Segment	*
Data Group	*	Far Data Group	*
Data Class	*	Far Data Class	*

Transfer

The **O**ptions | **T**ransfer command lets you add or delete programs in the ≡ menu. Once you've done so, you can run those programs without actually leaving Borland C++. You return to Borland C++ after you exit the program you transferred to. The **T**ransfer command displays this dialog box:

Figure 2.31
The Transfer dialog box



```
Program Titles
~GREP
~Turbo Assembler
Turbo ~Debugger
~Resource Compiler
~Import Librarian
```

The Transfer dialog box has two sections:

- the Program Titles list
- the Transfer buttons

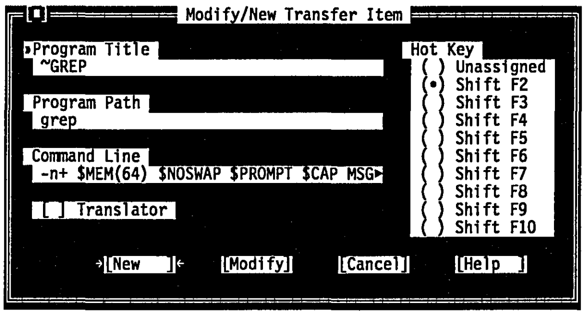
The Program Titles section lists short descriptions of programs that have been installed and are ready to execute. You might need to scroll the list box to see all the programs available.

The Transfer buttons let you edit and delete the names of programs you can transfer to, as well as cancel any changes you've made to the transfer list. There's also a Help button to get more information about using the transfer dialog box. Here's a rundown of the buttons.

The Edit button

Choose Edit to add or change the Program Titles list that appears in the ≡ menu. The Edit button displays the Modify/New Transfer Item dialog box.

If you're positioned on a transfer item when you select Edit, the input boxes in the Modify/New dialog box are automatically filled in; otherwise they're blank.



```
Program Title
 ~GREP
Program Path
 grep
Command Line
 -n+ $MEM(64) $NOSWAP $PROMPT $CAP MSG►
[ ] Translator

Hot Key
( ) Unassigned
(•) Shift F2
( ) Shift F3
( ) Shift F4
( ) Shift F5
( ) Shift F6
( ) Shift F7
( ) Shift F8
( ) Shift F9
( ) Shift F10
```

Figure 2.32
The Modify/New Transfer Item dialog box

Using the Modify/New dialog box, you take these steps to add a new file to the Transfer dialog box:

1. Type a short description of the program you're adding on the Program Title input box. (Note that when using a translator in a project, it must match the transfer title exactly.)

   Note that if you want your program to have a keyboard shortcut (like the *S* in the **S**ave command or the *t* in the Cut command), you should include a tilde (~) in the name. Whatever character follows the tilde appears in bold or in a special color in the ≡ menu, indicating that you can press that key to choose the program from the menu.
2. Tab to Program Path and enter the program name and optionally include the full path to the program. (If you don't enter an explicit path, only programs in the current directory or programs in your regular DOS path will be found.)

For a full description of these powerful macros, see the following section, "Transfer macros."

3. Tab to Command Line and type any parameters or macro commands you want passed to the program. Macro commands always start with a dollar sign ($) and are entered in uppercase. For example, if you enter `$CAP EDIT`, all output from the program will be redirected to a special Edit window in Borland C++.

This step is optional.

4. If you want to assign a hot key, tab to the Hot Key options and assign a shortcut to this program. Transfer shortcuts must be

Shift plus a function key. Keystrokes already assigned appear in the list but are unavailable.

5. Now click or choose the New button to add this program to the list.

To modify an existing transfer program, cursor to it in the Program Titles list of the Transfer dialog box and then choose Edit. After making the changes in the Modify/New Transfer dialog box, choose the Modify button.

`[ ] Translator`

The Translator check box lets you put the Transfer program into the Project File Translators list (the list you see when you choose **P**roject | **L**ocal Options). Check this option when you add a transfer program that is used to build part of your project.

The Delete button

The Delete button removes the currently selected program from the list and the ≡ menu.

Transfer macros

The IDE recognizes certain strings of characters called *transfer macros* in the parameter string of the Modify/New Transfer Item dialog box. There are three kinds of macros: state, file name, and instruction.

The transfer macros are listed alphabetically and described in more detail starting on page 77.

State macros are expanded according to the state of the IDE. The state macros are

$COL	$ERRNAME
$CONFIG	$INC
$DEF	$LIB
$ERRCOL	$LINE
$ERRLINE	$PRJNAME

File name macros are actually functions that take file names as arguments and return various parts of the file name. They allow you to build up new file name specifications from existing file names. For example, you can pass TDUMP a macro like this:

```
$DIR($EXENAME)$NAME($EDNAME).OBJ
```

This macro gives you the output directory path, the file name only in the active Edit window, and an explicit extension. If your current directory is C:\WORK, your output directory is TEST, and the active editor contains MYPROG.C, then TDUMP receives the parameter

```
C:\WORK\TEST\MYPROG.OBJ
```

The file name macros are

$DIR	$EXT()
$DRIVE()	$NAME()
$EDNAME	$OUTNAME
$EXENAME	

Instruction macros tell the IDE to perform some action or make some setting. The instruction macros are

$CAP EDIT	$PROMPT
$CAP MSG(*filter*)	$RC
$DEP()	$SAVE ALL
$IMPLIB	$SAVE CUR
$MEM(*kb to reserve*)	$SAVE PROMPT
$NOSWAP	$TASM

$CAP EDIT: This macro tells the IDE to redirect program output into a standard file. After the transfer program is completed, a new editor window is created, and the captured output is displayed. The captured output resides in a special Edit window titled Transfer Output.

For $CAP EDIT to work correctly, the transfer program must write to DOS standard output.

You can use any program that has line-oriented messages output (file and line number) with this macro.

$CAP MSG(filter): Captures program output into the Message window, using *filter* as a DOS filter for converting program output into Message window format.

We've provided four filters for this macro: GREP2MSG.EXE for GREP, IMPL2MSG.EXE for IMPLIB, RC2MSG.EXE for the Resource Compiler, and TASM2MSG.EXE for Turbo Assembler (TASM). We've included the source code for these filters so you can write your own filters for other transfer programs you install.

$COL: Column number of current editor. If the active window is not an editor, then the string is set to 0.

$CONFIG: 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.

TEML is a Pascal-like language that has many built-in primitive editor commands. Its use is documented online.

Use this macro with the Turbo Editor Macro Language (TEML) compiler. With it, you can edit the TEML script file in an editor and then invoke the Turbo Editor Macro Compiler (TEMC) to process the script. When the configuration file is reloaded, your new or modified editor commands will be in effect. When installing TEMC as a transfer item, use the following command line:

```
$EDNAME $CONFIG
```

This assumes the current Edit window contains the TEML script file to be processed.

$DEF: Pulls in the contents of the **O**ptions | **C**ompiler | **C**ode Generation "Defines" type-in box. Use this macro to specify define directives to an external translator.

This macro is only used by the project manager.

$DEP(): This macro provides the ability to automatically rebuild resources as part of a project make if one of the resource components has been modified. For example, let's say your Windows resource MYAPP1.RES for your application MYAPP1.EXE consists of the following files:

- MYAPP1.RC (the resource source file)
- MYAPP1.H (the header file for MYAPP1.EXE)
- MYAPP1.ICO (the icon for MYAPP1.EXE, included in MYAPP1.RC)
- MYAPP1.BMP (the bitmap used with MYAPP1.EXE, included in MYAPP1.RC)

To ensure that MYAPP1.RES gets recompiled any time you update one of the components, you'd add the following dependencies to the Options field in the **P**roject | **L**ocal Options dialog box for MYAPP1.RC:

```
$DEP(MYAPP1.H MYAPP1.ICO MYAPP1.BMP)
```

When you choose **C**ompile | **M**ake, the project manager scans for the $DEP macro and verifies that all explicit dependencies given are older than the resulting MYAPP1.RES file. If one or more aren't, the project manager recompiles MYAPP1.RC.

You can give explicit dependencies to any makeable project item. Just place the files you want the source to be dependent on in parentheses, separated by blanks, commas, or semicolons. If autodependency checking is on, explicit dependencies are checked after any autodependencies.

$DIR(): Directory of the file argument, full path.

$DRIVE(): Drive of the file argument, in the form *D*:.

$EDNAME: Complete file name of file in active editor. This is a null string if the active window is not an editor.

$ERRCOL: Column number of current error in file $ERRNAME. If there are no messages, then string is expanded to null string.

$ERRLINE: Line number of current error in file $ERRNAME. If there are no messages, then string is expanded to null string.

$ERRNAME: Complete file name of file referred to by the selected messages in the Message window. This is a null string if there are no messages or the currently selected message does not refer to a file.

$EXENAME: Program's file name (including output path), based on the project name or, if there is no project defined, then the name of the .EXE that would be produced from the active editor window. If the Windows DLL Linker option is selected, the file's extension will be .DLL.

$EXT(): Extension of the file argument; this includes the dot (for example, .CPP).

$IMPLIB: Executes IMPLIB.This macro expands to

```
$NOSWAP $CAP MSG(IMPL2MSG)
$DRIVE($EXENAME)$DIR($EXENAME)$NAME($EXENAME).LIB $EXENAME|def_name
```

If the Use DLL File Exports radio button is pushed (**O**ptions | **M**AKE | Generate Import Library), $EXENAME is part of the expansion. If the Use DEF file Exports radio button is pushed, the name of the DEF file in the project (represented by *def_name*) is used.

$INC: Pulls in the contents of the **O**ptions | **D**irectories | Include Directories type-in box.

$LIB: Pulls in the contents of the **O**ptions | **D**irectories | Library Directories type-in box.

$LINE: Line number of current editor. If the active window is not an editor, then the string is set to 0.

$MEM(*Kb to reserve*): This macro tells the IDE how much memory to try to give the transfer program. The IDE gives up as much memory as possible, to either the amount specified or the maximum available, whichever is smaller. You'll get an error if no memory is specified.

$NAME(): Name part of the file argument; does not include the dot.

$NOSWAP: This macro tells the IDE not to swap to the User Screen when running the program. It pops up a box that indicates which transfer program is running. Use this macro in conjunction with **$CAP**.

$OUTNAME: This macro expands to the path and file name that appear in the **P**roject | **L**ocal Options Output Path type-in box (in the active edit window). For example, if the project contains STARS.C, the default Output Path type-in is STARS.OBJ. So if STARS.C is in the active edit window, $OUTNAME expands to STARS.OBJ. If you've edited the type-in box so it says `..\MOON.XYZ`, $OUTNAME will expand to ..\MOON.XYZ. This macro is useful when you are specifying modules for your user-defined translators. For example, you could define a TLIB translator and set the command line to

```
TLIB MYLIB +$OUTNAME
```

which adds the object module of the file in the active edit window to the library MYLIB.

$PRJNAME: The current project file. Null string if no project is defined.

$PROMPT: This macro tells the IDE to display the expanded parameter string before calling the transfer program. The command line that will be passed is displayed in a dialog box. This allows you to change or add to the string before it is passed.The position of $PROMPT command in the command line determines what is shown in the dialog prompt box. You can place constant parameters in the command line by placing them before $PROMPT. For example, the **/c** in

```
  /c $PROMPT dir
```

is constant and doesn't show in the dialog box, but *dir* can be edited before the command is run.

$RC: This macro is predefined for use with the Resource Compiler. Since the Resource Compiler can be invoked for two separate reasons, $RC is expanded differently depending on whether you are compiling a .RC file into a .RES file or binding the .RES file to an executable file.

In any case, in order to change the behavior of the Resource Compiler

- when compiling an .RC file, change the command line in **P**roject | **L**ocal Options for the .RC file in the Project Manager
- when binding a .RES file to an .EXE or a DLL, change the options in **O**ptions | **T**ransfer | **R**esource Compiler

If you are compiling a .RC file into a .RES file, $RC is expanded like this:

```
$SAVE CUR $NOSWAP $CAP MSG(RC2MSG) -R -I$INC -FO $OUTNAME $EDNAME
```

If you are binding a .RES to an .EXE file, $RC is expanded like this:

```
$NOSWAP $CAP MSG(RC2MSG) res_name $EXENAME
```

The variable *res_name* is defined as one of the following, in this order:

1. If there is a file in the project with a .RES extension, *res_name* will be that file.
2. If there is no file with a .RES extension, and there *is* a file with a .RC extension, *res_name* is the name given by $OUTNAME for the .RC file.
3. If neither of the above apply (implying there are no resources), *res_name* is blank.

$SAVE ALL: This macro tells the IDE to save all modified files in all Edit windows that have been modified, without prompting.

$SAVE CUR: This macro tells the IDE to save the file in the current editor if it has been modified. This ensures that the invoked program will use the latest version of the source file.

$SAVE PROMPT: This macro tells the IDE to prompt when there are unsaved files in editor windows. You will be asked if you want to save any unsaved files.

$TASM: This macro is predefined for use with Turbo Assembler. It uses the TASM2MSG filter to trap TASM messages. **$TASM** is essentially shorthand for this:

```
$NOSWAP $SAVE CUR $CAP MSG(TASM2MSG) $EDNAME,$OUTNAME
```

WRITEMSG(*filename*): This macro copies the contents of the Message window to the specified ASCII file. The translator can parse the file and act on the messages so desired. For example, WRITEMSG(C:\MESSAGES.TXT) writes to the file MESSAGES.TXT on your root directory.

Running DOS commands

If you want to run DOS commands from within the integrated environment, you can set up a simple transfer macro that will let you do so. Just add this transfer item:

```
command /c $MEM(128) $PROMPT
```

When you invoke this transfer item, a dialog box appears and prompts you for DOS input. Since the $PROMPT command appears later in the string, the text *command /c* won't show up in the dialog's input box. This lets you just type `dir`, `chkdsk`, `del *.*`, or whatever DOS command you want to run.

Transfer memory settings

Different programs have different memory needs. For example, GREP can run in very little memory, where many popular editors require 200-300K to work well.

If you use the **$MEM()** macro, you can specify (on a program-by-program basis) how much memory the IDE should give to the transfer programs. The less memory you devote to a transfer program, the quicker the transfer to and from the program occurs.

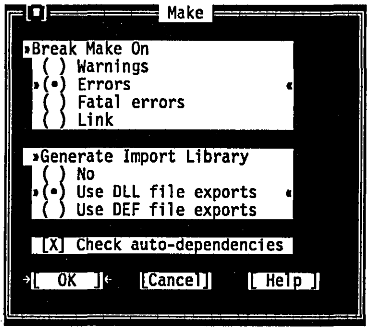
There may be some cases where the IDE cannot give up as much memory as you requested. When this happens, the IDE gives up as much as it can. There are certain states in the IDE that require more memory than others; for example, while debugging a program, the IDE will tie up more resources than when not debugging. Use **P**rogram Reset (*Ctrl-F2*) to free up debugging memory.

In those cases where you want the IDE to give up all its memory, give it a large number, like 640K. How much memory is actually given up is dependent on how much you have when you start Borland C++.

Make

The **O**ptions | **M**ake command displays a dialog box that lets you set conditions for project management. Here's what the dialog box looks like:

Figure 2.33
The Make dialog box



```
Make
Break Make On
  ( ) Warnings
  (•) Errors
  ( ) Fatal errors
  ( ) Link
Generate Import Library
  ( ) No
  (•) Use DLL file exports
  ( ) Use DEF file exports
[X] Check auto-dependencies
[ OK ]   [Cancel]   [ Help ]
```

```
Break Make On
  ( ) Warnings
  (•) Errors
  ( ) Fatal errors
  ( ) Link
```

Use the Break Make On radio buttons to set the condition that will stop the making of a project. The default is to stop after compiling a file with errors.

```
( )  No
(•)  Use DLL file exports
( )  Use DEF file exports
```

The Generate Import Library buttons control when and how IMPLIB is executed during the MAKE process. The Use DLL File Exports option generates an import library that consists of the exports in the DLL. The Use DEF File Exports generates an import library of exports in the DEF file. If either of these options is checked, MAKE invokes IMPLIB after the linker has created the DLL. This option controls how the transfer macro $IMPLIB gets expanded.

```
[X] Check Auto-dependencies
```

When the Check Auto-dependencies option is checked, the Project Manager automatically checks dependencies for every .OBJ file on disk that has a corresponding .C source file in the project list.

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 by both Borland C++ and the command-line version of Borland C++ when the source module is compiled. Then every file that was used to build the .OBJ file is checked for time and date 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 this option is off (unchecked), no such file checking is done.

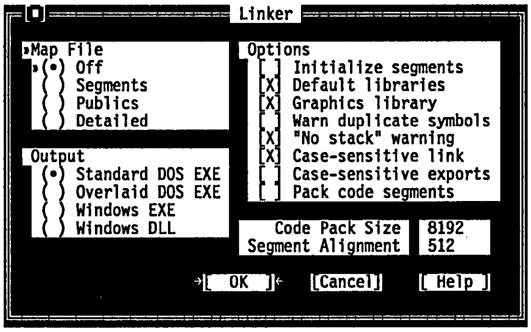
See the $DEP() transfer macro on page 78.

After the C source file is successfully compiled, the project file contains valid dependency infromation for that file. Once that valid information is in the project file, the Project Manager does the autodependency check using that information. This is much faster than reading each .OBJ file.

Linker

The **O**ptions | **L**inker command lets you make several settings that affect linking. The **L**inker command opens this dialog box:

Figure 2.34
The Linker dialog box



```
[■]═══════════════════ Linker ═══════════════════
 Map File                 Options
  (•) Off                  [ ] Initialize segments
  ( ) Segments             [X] Default libraries
  ( ) Publics              [X] Graphics library
  ( ) Detailed             [ ] Warn duplicate symbols
                           [X] "No stack" warning
 Output                    [X] Case-sensitive link
  (•) Standard DOS EXE     [ ] Case-sensitive exports
  ( ) Overlaid DOS EXE     [ ] Pack code segments
  ( ) Windows EXE
  ( ) Windows DLL             Code Pack Size   8192
                           Segment Alignment   512

                 [  OK  ]    [Cancel]    [ Help ]
```

This dialog box has several check boxes and radio buttons. The following sections contain short descriptions of what each does.

```
Map File
 (•) Off
 ( ) Segments
 ( ) Publics
 ( ) Detailed
```

Use the Map File radio buttons to choose the type of map file to be produced. For settings other than Off, the map file is placed in the output directory defined in the **O**ptions | **D**irectories dialog box. The default setting for the map file is Off.

```
Output
 (•) Standard DOS EXE
 ( ) Overlaid DOS EXE
 ( ) Windows EXE
 ( ) Windows DLL
```

Use these radio buttons to set your application type. Standard DOS EXE produces a normal executable that runs under DOS. Overlaid DOS EXE produces an executable that is capable of being overlaid. Windows EXE produces a Windows application, while Windows DLL produces a Windows dynamic link library.

```
[ ] Initialize segments
```

If checked, Initialize Segments tells the linker to initialize uninitialized segments. (This is normally not needed and will make your .EXE files larger.)

```
[ ] Default libraries
```

When you're linking with modules created by a compiler other than Borland C++, the other compiler may have placed a list of default libraries in the object file.

If the Default Libraries option is checked, the linker tries to find any undefined routines in these libraries as well as in the default libraries supplied by Borland C++. If this option is off (unchecked), the linker searches only the default libraries supplied by Borland C++ and ignores any defaults in .OBJ files.

```
[X] Graphics library
```

The Graphics Library option controls the automatic searching of the BGI graphics library. When this option is checked (the default), it's possible to build and run single-file graphics programs without using a project file. Unchecking this option speeds up the link step a bit because the linker doesn't have to search in the BGI graphics library file.

Note: You can uncheck this option and still build programs that use BGI graphics, provided you add the name of the BGI graphics library (GRAPHICS.LIB) to your project list.

```
[X] Warn duplicate symbols
```

The Warn Duplicate Symbols option affects whether the linker warns you of previously encountered symbols in .LIB files.

```
[ ] "No stack" warning
```

The "No Stack" Warning option affects whether the linker generates the "No stack" message. It's normal for a program generated under the tiny model to display this message if the message is not turned off.

```
[X] Case-sensitive Link
```

The Case-Sensitive Link option affects whether the linker is case-sensitive. Normally, this option should be checked, since C and C++ are both case-sensitive languages.

```
[ ] Case-sensitive exports
```

By default, the linker ignores case with the names in the IMPORTS and EXPORTS sections of the module definition file. If you want the linker be case-sensitive in regard to these names, check this option. This option is probably only useful when you are trying to export non-callback functions from DLLs—as in exported C++ member functions. This option isn't necessary for normal Windows callback functions (declared FAR PASCAL).

```
[ ] Pack code segments
```

This option applies only to Windows applications and DLLs. When this option is checked, the linker tries to minimize the number of code segments by packing multiple code segments together; typically, this will improve performance. This option will never create segments greater than 64K.

```
   Code Pack Size   8192
Segment Alignment   512
```

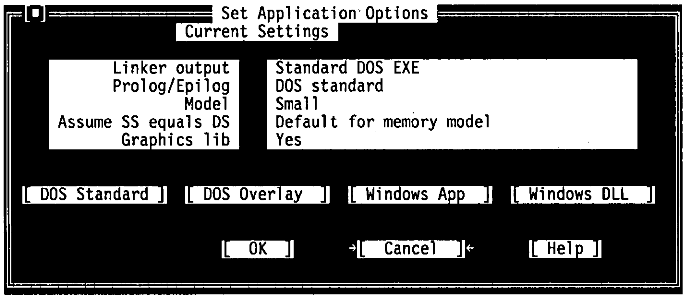
You can change the default code packing size to anything between 1 and 65,536 with Code Pack Size. See page 247 for a more in-depth discussion of desirable sizes.

With Segment Alignment, you can set the segment alignment. Note that the alignment factor will be automatically rounded up to the nearest power of two (meaning that if you enter 650, it will be rounded up to 1,024). The possible numbers you can enter must fall in the range of 2 to 65,535.

The Set Application Options dialog box

This dialog box provides the easiest and safest way to set up compilation and linking for a DOS or Windows executable. To use this dialog box, simply push one of the buttons. Borland C++ will verify and, if necessary, changes some of the settings in the Code Generation, Entry/Exit Code Generation, and Linker dialog boxes. See page 65 (Entry/Exit Code) for detailed information on the code generated. Use this dialog box for initial setup only.

Figure 2.35
Set Application Options



Linker output	Standard DOS EXE
Prolog/Epilog	DOS standard
Model	Small
Assume SS equals DS	Default for memory model
Graphics lib	Yes

The standard options for applications and libraries each accomplish a set of tasks. You can choose only one button at a time. The current settings fields are updated when you press the button.

DOS Standard:

- pushes the Small memory model radio button in the Code Generation dialog box
- unchecks Assume SS not equal DS in the Code Generation dialog box
- pushes the DOS Standard radio button in the Entry/Exit Code Generation dialog box
- pushes the Standard DOS .EXE radio button in the Linker dialog box
- checks the Graphics Library option in the Linker dialog box

DOS Overlay:

- pushes the Medium memory model button in the Code Generation dialog box
- unchecks Assume SS not equal DS in the Code Generation dialog box

- pushes the DOS Overlay button in the Entry/Exit Code Generation dialog box
- pushes the Overlaid DOS .EXE button in the Linker dialog box
- checks the Graphics Library option in the Linker dialog box

Windows App:

- pushes the Small memory model button in the Code Generation dialog box
- unchecks Assume SS not equal DS in the Code Generation dialog box
- pushes the Windows All Functions Exportable button in the Entry/Exit Code Generation dialog box
- pushes the Windows .EXE button in the Linker dialog box
- unchecks the Graphics Library option in the Linker dialog box

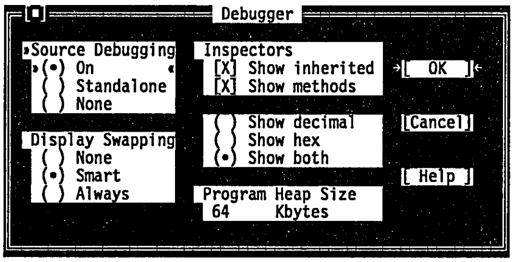
Windows DLL:

- pushes the Compact memory model button in the Code Generation dialog box
- checks Assume SS not equal DS in the Code Generation dialog box
- pushes the Windows DLL All Functions Exportable button in the Entry/Exit Code Generation dialog box
- unchecks the Graphics Library option in the Linker dialog box

Debugger

The **O**ptions | De**b**ugger command lets you make several settings affecting the integrated debugger. This command opens this dialog box:

Figure 2.36
The Debugger dialog box


```
Debugger
Source Debugging        Inspectors
 (•) On                  [X] Show inherited      [  OK  ]
 ( ) Standalone          [X] Show methods
 ( ) None
                         ( ) Show decimal        [Cancel]
Display Swapping         ( ) Show hex
 ( ) None                (•) Show both
 (•) Smart                                       [ Help ]
 ( ) Always             Program Heap Size
                         64     Kbytes
```

The following sections describe the contents of this box.

```
Source Debugging
 (•) On
 ( ) Standalone
 ( ) None
```

The Source Debugging radio buttons determine whether debugging information is included in the executable file and how the .EXE is run under Borland C++.

Programs linked with this option set to On (the default) can be debugged with either the integrated debugger or the standalone Turbo Debugger. Switch this back to On when you want to debug in the IDE.

If you set this option to Standalone, programs can be debugged only with Turbo Debugger, although they can still be run in Borland C++.

If you set this option to None, programs cannot be debugged with either debugger, because no debugging information has been placed in the .EXE file.

```
Display Swapping
 ( ) None
 (•) Smart
 ( ) Always
```

The Display Swapping radio buttons let you set when the integrated debugger will change display windows while running a program.

If you set Display Swapping to None, the debugger does not swap the screen at all. You should only use this setting for debugging sections of code that you're certain do not output to the screen.

When you run your program in debug mode with the default setting of Smart, the debugger looks at the code being executed to see whether it will generate output to the screen. If the code does output to the screen (or if it calls a function), the screen is swapped from the IDE screen to the User Screen long enough for output to be displayed, then is swapped back. Otherwise, no swapping occurs.

➡ Be aware of the following with smart swapping:

- It swaps on any function call, even if the function does no screen output.
- In some situations, the IDE screen might be modified without being swapped; for example, if a timer interrupt routine writes to the screen.

If you set Display Swapping to Always, the debugger swaps screens every time a statement executes. You should choose this setting any time the IDE screen is likely to be overwritten by your running program.

Note If you're debugging in dual monitor mode (that is, you used the Borland C++ command-line **/d** option), you can see your program's output on one monitor and the Borland C++ screen on the other. In this case, Borland C++ never swaps screens and the Display Swapping setting has no effect.

```
Inspectors
 [X] Show inherited
 [X] Show methods

 ( ) Show decimal
 ( ) Show hex
 (•) Show both
```

In the Inspectors checkboxes, when Show Inherited is checked, it tells the integrated debugger to display all member functions and methods—whether they are defined within the inspected class or inherited from a base class. When this option is not checked, only those fields defined in the type of the inspected object are displayed.

When checked, the Show Methods option tells the integrated debugger to display member functions when you inspect a class.

Check the Show Decimal, Show Hex, or Show Both radio buttons when you want to control how the values in inspectors are displayed. Show both is on by default.

```
Program Heap Size
  64    Kbytes
```

You can use the Program Heap Size input box to input how much memory Borland C++ should assign a program when you debug it. The actual amount of memory that Borland C++ tries to give to the program is equal to the size of the executable image plus the amount you specify here.

Usually, it's only meaningful to increase heap size when working with large data models.

The default value for the program heap size is 64 Kbytes. You may want to increase this value if your program uses dynamically allocated objects.

Directories

The **O**ptions | **D**irectories command lets you tell Borland C++ where to find the files it needs to compile, link, and output binary and map files. This command opens the following dialog box containing three input boxes:

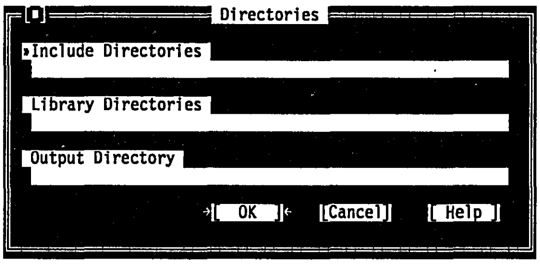


Figure 2.37
The Directories dialog box

Here is what each input box is for:

- The Include Directories input box specifies the directory that contains your include files. Standard include files are those given in angle brackets (<>) in an **#include** statement (for example, **#include** <*myfile.h*>). These directories are also searched for quoted Includes not found in the current directory. Multiple directory names are allowed, separated by semicolons.

- The Library Directories input box specifies the directories that contain your Borland C++ startup object files (C0?.OBJ) and run-time library files (.LIB files) and any other libraries that your project may use. Multiple directory names are allowed, separated by semicolons.
- The Output Directory input box specifies the directory that stores your .OBJ, .EXE, and .MAP files. Borland C++ looks for that directory when doing a make or run, and to check dates and times of .OBJs and .EXEs. If the entry is blank, the files are stored in the current directory.

Use the following guidelines when entering directories in these input boxes:

- 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\LIB;C:\C\MYLIBS;A:\BORLANDC\MATHLIBS;A:..\VIDLIBS
```

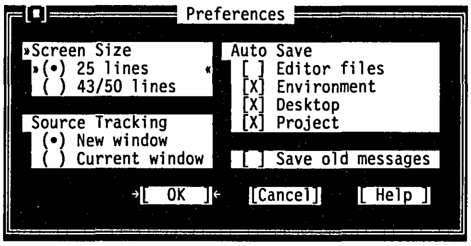
Environment

The **O**ptions | **E**nvironment command lets you make environment-wide settings. This command opens a menu that lets you choose settings from **P**references, **E**ditor, **M**ouse, and **D**esktop.

Preferences

Here's what the Preferences dialog box looks like:

Figure 2.38
The Preferences dialog box



```
Preferences
»Screen Size              Auto Save
 »(•) 25 lines «           [ ] Editor files
  ( ) 43/50 lines          [X] Environment
                           [X] Desktop
 Source Tracking           [X] Project
  (•) New window
  ( ) Current window       [ ] Save old messages

        [ OK ]      [Cancel]      [ Help ]
```

```
Screen Size
 (•) 25 lines
 ( ) 43/50 lines
```

The Screen Size radio buttons let you specify whether your IDE screen is displayed in 25 lines or 43/50 lines. One or both of these buttons will be available, depending on the type of video adapter in your PC.

When set to 25 lines (the default), Borland C++ uses 25 lines and 80 columns. This is the only screen size available to systems with a monochrome display or Color Graphics Adapter (CGA).

If your PC has EGA or VGA, you can set this option to 43/50 lines. The IDE is displayed in 43 lines by 80 columns if you have an EGA, or 50 lines by 80 columns if you have a VGA.

```
Source Tracking
 ( ) New window
 (•) Current window
```

When stepping source or viewing the source from the Message window, the IDE opens a new window whenever it encounters a file that is not already loaded. Selecting Current Window causes the IDE to replace the contents of the topmost Edit window with the new file instead of opening a new Edit window.

```
Auto Save
 [ ] Editor Files
 [X] Environment
 [X] Desktop
 [X] Project
```

If Editor Files is checked in the Auto Save options, and if the file has been modified since the last time you saved it, Borland C++ automatically saves the source file in the Edit window whenever you choose the **Run | Run** (or any debug/run command) or **File | DOS** Shell command.

When the Environment option is checked, all the settings you made in this dialog box will be saved automatically when you exit Borland C++.

When Desktop is checked, Borland C++ controls whether your desktop is saved on exit and whether it's restored when you return to Borland C++.

When the Project option is checked, Borland C++ saves all your project, auto-dependency, and module settings on exit and restores them when you return to Borland C++.

```
[ ] Save Old Messages
```

When Save Old Messages is checked, Borland C++ saves the error messages currently in the Message window, appending any messages from further compiles to the window. When a file is compiled, any messages for that file are removed from the Message window and new messages are added to the end. Messages are not saved from one session to the next. When you uncheck this box, Borland C++ automatically clears messages before a compile, a make, or a transfer that uses the Message window.

Editor

If you choose **E**ditor from the **E**nvironment menu, these are the options you can pick from:

```
Editor Options
 [X] Create backup files
 [X] Insert mode
 [X] Autoindent mode
 [X] Use tab character
 [X] Optimal fill
 [X] Backspace unindents
 [X] Cursor through tabs
 [ ] Group undo
 [X] Persisten blocks
```

- When Create Backup Files is checked (the default), Borland C++ automatically creates a backup of the source file in the Edit window when you choose **F**ile | **S**ave and gives the backup file the extension .BAK.
- When Insert Mode is not checked, any text you type into Edit windows overwrites existing text. When the option is checked, text you type is inserted (pushed to the right). Pressing *Ins* toggles Insert mode when you're working in an Edit window.
- When Autoindent Mode is checked, pressing *Enter* in an Edit window positions the cursor under the first nonblank character in the preceding nonblank line. This can be a great aid in keeping your program code more readable.
- When Use Tab Character is checked, Borland C++ inserts a true tab character (ASCII 9) when you press *Tab*. When this option is not checked, Borland C++ replaces tabs with spaces. If there are any lines with characters on them prior to the current line, the cursor is positioned at the first corresponding column of characters following the next whitespace found. If there is no "next" whitespace, the cursor is positioned at the end of the line. After the end of the line, each *Tab* press is determined by the Tab Size setting.
- When you check Optimal Fill, Borland C++ begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters than when Optimal Fill is not checked.
- When Backspace Unindents is checked (which is the default) and the cursor is on a blank line or the first non-blank character

of a line, the *Backspace* key aligns (outdents) the line to the previous indentation level.

- When you check Cursor Through Tabs, the arrow keys will move the cursor to the middle of tabs; otherwise the cursor jumps several columns when cursoring over a tab.
- When Group Undo is unchecked, choosing **Edit | Undo** reverses the effect of a single editor command or keystroke. For example, if you type `ABC`, it will take three **Undo** commands to delete *C*, then *B*, then *A*.

  If Group Undo is checked, **Undo** reverses the effects of the previous command and all immediately preceding commands of the same type. The types of commands that are grouped are insertions, deletions, overwrites, and cursor movements. For example, if you type ABC, one **Undo** command deletes ABC.

  For the purpose of grouping, inserting a carriage return is considered an insertion followed by a cursor movement. For example, if you press *Enter*, then type ABC, choosing **Undo** once will delete the ABC, and choosing **Undo** again will move the cursor to the new carriage return. Choosing **Edit | Redo** at that point would move the cursor to the following line. Another **Redo** would insert ABC. (See page 34 for more information about **Undo** and **Redo**.)
- When this option is checked (the default), marked blocks behave as they always have in Borland's C and C++ products. With this option unchecked, marked blocks behave differently in these instances:

  1. Pressing the *Del* key or the *Backspace* key clears the entire selected text.
  2. Inserting text (pressing a character, pasting from clipboard) replaces the entire selected text with the inserted text.
  3. Moving the cursor out of the block de-selects the text.

`Tab Size 8`

If you check Use Tab Character in this dialog box and press *Tab*, Borland C++ inserts a tab character in the file and the cursor moves to the next tab stop. The Tab Size input box allows you to dictate how many characters to move for each tab stop. Legal values are 2 through 16; the default is 8.

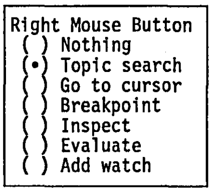
To change the way tabs are displayed in a file, just change the tab size value to the size you prefer. Borland C++ redisplays all tabs in that file in the size you chose. You can save this new tab size in your configuration file by choosing **O**ptions | **S**ave Options.

```
Default Extension  CPP
```

The Default Extension input box lets you tell Borland C++ which extension to use as the default when compiling and loading your source code. Changing this extension doesn't affect the history lists in the current desktop.

Mouse

When you choose **M**ouse from the **E**nvironment menu, the Mouse Options dialog box is displayed, which contains all the settings for your mouse. These are the options available to you:

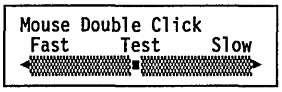


```
Right Mouse Button
 ( ) Nothing
 (•) Topic search
 ( ) Go to cursor
 ( ) Breakpoint
 ( ) Inspect
 ( ) Evaluate
 ( ) Add watch
```

The Right Mouse Button radio buttons determine the effect of pressing the right button of the mouse (or the left button, if the reverse mouse buttons option is checked). Topic Search is the default.

Here's a list of what the right button would do if you choose something other than Nothing:

Topic Search	Same as **H**elp | **T**opic Search
Go to Cursor	Same as **R**un | **G**o To Cursor
Breakpoint	Same as **D**ebug | **T**oggle Breakpoint
Inspect	Same as **D**ebug | **I**nspect
Evaluate	Same as **D**ebug | **E**valuate
Add Watch	Same as **D**ebug | **W**atches | **A**dd Watch



```
 Mouse Double Click
  Fast      Test      Slow
```

In the Mouse Double Click box, you can change the slider control bar to adjust the double-click speed of your mouse by using the arrow keys.

Moving the scroll box closer to Fast means Borland C++ requires a shorter time between clicks to recognize a double click. Moving the scroll box closer to Slow means Borland C++ will still recognize a double click even if you wait longer between clicks.

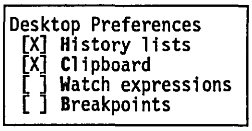
If you want to experiment with different settings, you can double-click the Test button above the scroll bar. When you successfully double-click the bar it becomes highlighted.

```
 [ ] Reverse Mouse Buttons
```

When Reverse Mouse Buttons is checked, the active button on your mouse is the rightmost one instead of the leftmost. Note, however, that the buttons won't actually be switched until you choose the OK button.

Depending on how you hold your mouse and whether you're right- or left-handed, the right mouse button might be more comfortable to use than the left.

Desktop



```
Desktop Preferences
 [X] History lists
 [X] Clipboard
 [ ] Watch expressions
 [ ] Breakpoints
```

The **D**esktop dialog box lets you set whether history lists, the contents of the Clipboard, watch expressions, and breakpoints are saved across sessions. History lists and the contents of the Clipboard are saved by default; because watch expressions and breakpoints may not be meaningful across sessions, they are not saved by default. You can change these defaults by unchecking or checking the respective options.

Save

The **O**ptions | **S**ave command brings up a dialog box that lets you save settings that you've made in both the **F**ind and **R**eplace dialog boxes (off the **S**earch menu) and in the **O**ptions menu (which includes all the dialog boxes that are part of those commands) for IDE, Desktop, and Project items. Options are stored in three files, which represent each of these categories. If it doesn't find the files, Borland C++ looks in the Executable directory (from which BC.EXE or BCX.EXE is run) for the same file.

Window menu

The **W**indow menu contains window management commands. Most of the windows you open from this menu have all the standard window elements like scroll bars, a close box, and zoom boxes. Refer to page 13 for information on these elements and how to use them.

At the bottom of the **W**indow menu, the **W**indow | **L**ist command appears. Choose this command for a list of all open windows as well as recently closed ones. (A recently closed window appears with *closed* before it; choose it to reopen it.)

Size/Move

Ctrl F5

Choose **W**indow | **S**ize/Move to change the size or position of the active window.

When you choose this command, the active window moves in response to the arrow keys. When the window is where you want it, press *Enter*. You can also move a window by dragging its title bar.

If you press *Shift* while you use the arrow keys, you can change the size of the window. When it's the size you want it, press *Enter*. If a window has a resize corner, you can drag that corner or any other corner to resize it.

Zoom

F5

Choose **Window | Zoom** to resize the active window to the maximum size. If the window is already zoomed to the max, you can choose this command again to restore it to its previous size. You can also double-click anywhere on the top line (except where an icon appears) of a window to zoom or unzoom it.

Tile

Choose **Window | Tile** to tile all your open windows.

Cascade

Choose **Window | Cascade** to stack all open windows.

Next

F6

Choose **Window | Next** to make the next window active, which makes it the topmost open window.

Close

Alt F3

Choose **Window | Close** to close the active window. You can also click the close box in the upper left corner to close a window.

Message

Choose **Window | Message** to open the Message window and make it active. The Message window displays error and warning messages, which you can use for reference, or you can select them and have the corresponding location be highlighted in the Edit window. When a message refers to a file that is not currently loaded, you can press the *Spacebar* to load that file. You can also display transfer program output in this window.

When an error is selected in the Message window, press *Enter* to show the location of the error in the Edit window and make the Edit window active at the point of error.

To close the window, click its close box or choose **W**indow | **C**lose.

Output

Choose **W**indow | **O**utput to open the Output window and make it active. The Output window displays text from any DOS command-line text and any text generated from your program (no graphics).

The Output window is handy while debugging because you can view your source code, variables, and output all at once. This is especially useful when you've set the **O**ptions | **E**nvironment dialog box to a 43/50 line display and you are running a standard 25-line mode program. In that case, you can see almost all of the program output and still have plenty of lines to view your source code and variables.

If you would rather see your program's text on the full screen—or if your program generates graphics—choose the **W**indow | **U**ser Screen command instead.

To close the window, click its close box or choose **W**indow | **C**lose.

Watch

Choose **W**indow | **W**atch to open the Watch window and make it active. The Watch window displays expressions and their changing values so you can keep an eye on how your program evaluates key values.

You use the commands in the **D**ebug | **W**atches pop-up menu to add or remove watches from this window. Refer to the section on this menu for information on how to use the Watch window (page 54).

To close the window, click its close box or choose **W**indow | **C**lose.

User Screen

Alt F5

Choose **W**indow | **U**ser Screen to view your program's full-screen output. If you would rather see your program output in a Borland C++ window, choose the **W**indow | **O**utput command instead. Clicking or pressing any key returns you to the IDE.

Register

Choose **W**indow | **R**egister to open the Register window and make it active.

The Register window displays CPU registers and is used when debugging inline ASM and TASM modules in your project.

To close the window, click its close box or choose **W**indow | **C**lose.

Project

Choose **W**indow | **P**roject to open the Project window, which lets you view files that you're using to create your program.

Project Notes

Choose **W**indow | Project **N**otes to write down any details, make to-do lists, or list any other information about your project files.

List

Choose **W**indow | **L**ist to get a list of all the windows you've opened. The list contains the names of all files that are currently open as well as any of the last eight files you've opened in an Edit window but have since closed. A recently closed file appears in the list prefixed with the word *closed.*

When you choose an already open file from the list, Borland C++ brings the window to the front and makes it active. When you choose a closed file from the list, Borland C++ reopens the file in an Edit window the same size and location as when the window was closed. The cursor is positioned at its last location.

Alt 0

Press *Alt-0* to pop up a complete list of all open windows and all Edit windows you've closed. For a full rundown of how to manage windows, see page 15.

Help menu

The **H**elp menu gives you access to online help in a special window. There is help information on virtually all aspects of the

IDE and Borland C++. (Also, one-line menu and dialog box hints appear on the status line whenever you select a command.)

To open the Help window, do one of these actions:

F1

- Press *F1* at any time (including from any dialog box or when any menu command is selected).
- When an Edit window is active and the cursor is positioned on a word, press *Ctrl-F1* to get language help.
- Click Help whenever it appears on the status line or in a dialog box.

To close the Help window, press *Esc*, click the close box, or choose **W**indow | **C**lose. You can keep the Help window onscreen while you work in another window unless you opened the Help window from a dialog box or pressed *F1* when a menu command was selected. (If you press *F6* or click on another window while you're in Help, the Help window remains onscreen.)

When getting help in a dialog box or menu, you cannot resize the window or copy to the clipboard. In this instance, Tab takes you to dialog box controls, not the next keyword.

Help screens often contain *keywords* (highlighted text) that you can choose to get more information. Press *Tab* to move to any keyword; press *Enter* to get more detailed help. (As an alternative, move the cursor to the highlighted keyword and press *Enter*. With a mouse, you can double-click any keyword to open the help text for that item.

You can also cursor around the Help screen and press *Ctrl-F1* on *any* word to get help. If the word is not found, an incremental search is done in the index and the closest match displayed.

When the Help window is active, you can copy from the window and paste that text into an Edit window. You do this just the same as you would in an Edit window: Select the text first (using *Shift*→ , Left arrow, Up arrow, Down arrow), choose **E**dit | **C**opy, move to an Edit window, then choose **E**dit | **P**aste.

To select text in the Help window, drag across the desired text or, when positioned at the start of the block, press *Shift*→, ←, ↑, ↓ to mark a block.

You can also copy preselected program examples from help screens by choosing the **E**dit | Copy **E**xample command.

Contents

The **H**elp | **C**ontents command opens the Help window with the main table of contents displayed. From this window, you can branch to any other part of the help system.

F1 You can get help on Help by pressing *F1* when the Help window is active. You can also reach this screen by clicking on the status line.

Index

The **H**elp | **I**ndex command opens a dialog box displaying a full list of help keywords (the special highlighted text in help screens that let you quickly move to a related screen).

You can scroll the list or you can incrementally search it by pressing letters from the keyboard. For example, to see what's available under "printing," you can type `p r i`. When you type `p`, the cursor jumps to the first keyword that starts with *p*. When you then type `r`, the cursor then moves to the first keyword that starts with *pr*. When you then type *i*, the cursor moves to the first keyword that starts with *pri*, and so on.

You can also tab to a keyword to select it.

When you find a keyword that interests you, choose it by cursoring to it and pressing *Enter*. (You can also double-click it.)

Topic Search

Ctrl F1 The **H**elp | **T**opic Search command displays language help on the currently selected item.

To get language help, position the cursor on an item in an Edit window and choose **T**opic Search. You can get help on things like function names (**printf**, for example), header files, reserved words, and so on. If an item is not in the help system, the help index displays the closest match.

Previous Topic

Alt F1 The **H**elp | **P**revious Topic command opens the Help window and redisplays the text you last viewed.

Borland C++ lets you back up through 20 previous help screens. You can also click on the status line to view the last help screen displayed.

Help on Help

F1

The **Help** | **Help** on Help command opens up a text screen that explains how to use the Borland C++ help system. If you're already in help, you can bring up this screen by pressing *F1*.

C H A P T E R

3

Building a Windows application

This chapter explains how to use Borland C++ to build Windows applications or dynamic link libraries (DLLs). This chapter does *not* explain the intricacies of designing Windows applications, nor does it teach you how to program under Windows—these topics are far beyond the scope of this chapter or this book.

The basic process

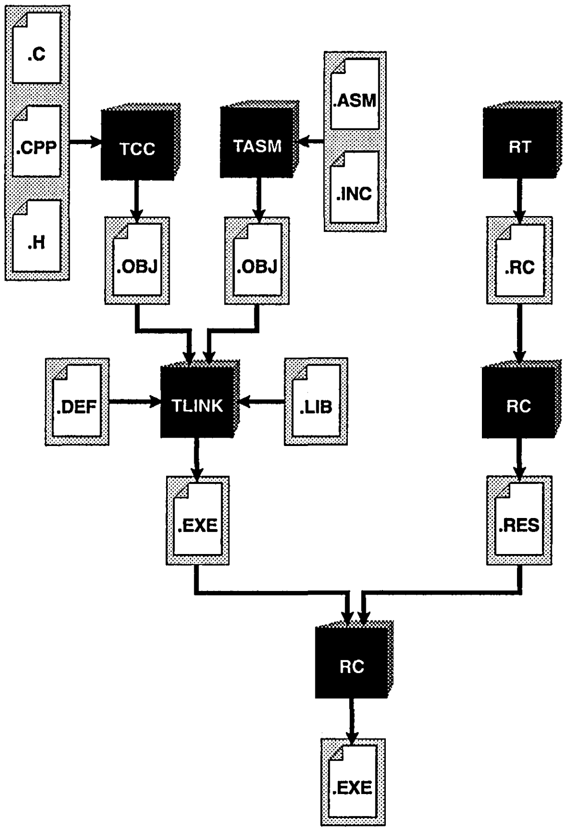
Compiling and linking a module for Windows is basically the same as it is for DOS. The compiler first generates an object file (which differs from a DOS compilation primarily in the special Windows prolog and epilog code that wraps each function). The prolog and epilog code varies depending on which Windows compilation options are used; these options are described later.

To create a Windows module for the memory model you are compiling under, the linker links the object files with the appropriate Borland C++ startup code, various libraries, and the module definition file.

Finally, either the IDE, the makefile, or the programmer invokes the Resource Compiler to bind the resources to the module. Figure 3.1 illustrates the entire process.

The next section, "Compiling and linking with the IDE," gives you a quick example of how to compile, link, and run a Windows program in the Borland C++ IDE. If you normally compile and link from the command line or from a makefile, then you should read "Compiling and linking from the command line," starting on page 107.

Figure 3.1
Compiling and linking a Windows program



Compiling and linking within the IDE

You can find complete descriptions of the various IDE commands and options in Chapter 2.

By way of example, you'll be producing a simple Windows application called WHELLO, which creates a window and writes a text message to that window. WHELLO.EXE is produced by compiling and linking the following three files:

- WHELLO.CPP, the C++ source file
- WHELLO.RC, the resource file
- WHELLO.DEF, the module definition file

Understanding resource files

Windows applications typically use *resources*, which can be icons, dialog boxes, fonts, cursors, and bitmaps. These resources can be created by the Resource Toolkit and are defined in a file called a resource file. For this application, the resource file is WHELLO.RC.

.RC resource files are source files, also called resource script files. Before an .RC file can be added to an executable, the .RC file must first be compiled by the Resource Compiler into a binary format; compilation creates a .RES file. For instance, compiling WHELLO.RC with the Resource Compiler creates WHELLO.RES. The Resource Compiler is also used to bind .RES resource files to an executable file.

To build a final Windows application, complete with resources, you need to invoke the Resource Compiler in order to bind the .RES file to the .EXE file. The Resource Compiler does three things:

1. It compiles .RC files to .RES files.
2. It binds the .RES file to the compiled module (.EXE or .DLL).
3. It marks the .EXE or .DLL as Windows-compatible.

Understanding module definition files

Module definition files are described in detail on page 250.

The module definition file WHELLO.DEF provides information to the linker about the contents and system requirements of a Windows application. Because TLINK and the built-in linker have other ways of finding out the information contained in the module definition, module definition files are not required for Borland C++'s linker to create a Windows application, although one is included here for the sake of example.

Compiling and linking WHELLO

Here's how you turn these three files into a Windows application:

1. If you haven't done so already, go to the \BORLANDC\ EXAMPLES directory and start the Borland C++ IDE by typing `BCX` (protected mode version) or `BC` from the DOS

command line. If you are already in the IDE, change to the \BORLANDC\EXAMPLES directory with **F**ile | **C**hange dir.

2. Choose **P**roject | **O**pen Project. In the Project Name box, type WHELLO.PRJ. Press *Enter* or click OK to open a new project with the name WHELLO.
3. Choose **P**roject | **A**dd item and type `whello.*` in the Name box, so that you'll get a list of all the WHELLO files.
4. Add the three files WHELLO.CPP, WHELLO.RC, and WHELLO.DEF for the application. Close the Project dialog box after you've added the three files.
5. Choose **O**ptions | **A**pplication to open the Set Application Options dialog box. Choose Windows App. The information pane at the top of the dialog box changes. Each of the four buttons at the bottom of the dialog checks and sets several other options in the IDE (see Chapter 2, page 86 for details).
6. Choose **C**ompile | **B**uild all to build the project.
7. Exit the IDE by pressing *Alt-X* or choosing **F**ile | **Q**uit.
8. From the DOS command line, type

   ```
   win whello
   ```

   DOS will load Windows, which will itself run the WHELLO application.

That's all there is to building and running a Windows application with Borland C++. You can generalize this process into the following checklist:

1. Create a project.
2. Add the source files, resource files, import libraries (if necessary), and the module definition file (if necessary) to the project.
3. Set up the compilation and link environment with the Set Application Options dialog box, or with a combination of other settings and options.
4. Build the project.
5. Run the application under Windows.

Setting compile and link options

The bulk of the setup in this example is accomplished by the Set Application Options dialog box. The action buttons in this dialog box check or set various other options in other dialog boxes. Borland C++ makes it easy for you to change the settings that control compilation and linkage of your programs, so you'll want to familiarize yourself with the following dialog boxes (all described in full in Chapter 2):

- The Code Generation dialog box sets such things as the memory model, tells the compiler to use pre-compiled headers, and more. Choose **O**ptions | **C**ompiler | **C**ode Generation to see this dialog box.
- The Entry/Exit Code Generation dialog box sets Borland C++ compiler options for prolog and epilog code generation, and export options. Choose **O**ptions | **C**ompiler | **E**ntry/Exit Code and browse through the contents of this dialog box.
- The Make dialog box (**O**ptions | **M**ake). The Generate Import Library options allow you to create an import library for a DLL. An import library makes it possible to declare all of the functions in a DLL as imports to another module without using a module definition file (see Chapter 7, page 192).
- The Linker dialog box (**O**ptions | **L**inker) sets options for the type of output you want from the linker—such as a standard DOS .EXE, an overlaid DOS .EXE, a Windows .EXE, or a Windows DLL—as well as a number of other linker options.

Compiling and linking from the command line

If you know how to compile and link a C++ or C program for DOS, then you already know almost all you need to know to do the same thing for Windows. You'll need three files to compile and link the example application:

- WHELLO.CPP, the C++ source code
- WHELLO.DEF, the module definition file
- WHELLO.RC, the resource file

Compiling from the command line

To compile and link WHELLO.CPP for a Windows application, type

```
BCC -W whello.cpp
```

Given this command line, Borland C++ compiles WHELLO.CPP into WHELLO.OBJ, then links in the correct libraries and startup code automatically. To suppress the link phase, add the **-c** option to the command line. To include debugging information, add the **–v** option.

The **–W** option tells the command-line compiler that you want a Windows application. There are other Windows options (of the form **–W*xxx***) that give the compiler more specific instructions about the compilation and code generation of a Windows application (for instance, **–WD** to create a DLL). You can find detailed descriptions of all the command-line options in Chapter 6.

Once the WHELLO application is compiled and linked, the only thing left to do is add the resources. First, compile the WHELLO.RC file with the command

```
rc -r whello.rc
```

This produces a WHELLO.RES file (-r instructs the Resource Compiler to *not* add the result to the executable of the same name). Now, invoke the Resource Compiler again to add the binary resource file to the executable.

```
rc whello.res whello.exe
```

Actually, the Resource Compiler makes it easier than we've shown here, because it can compile an .RC file into a .RES file and then add it to the executable all in one step. Furthermore, if the executable file has the same first name as the resource file, then you don't need to specify the executable file on the command line at all. So, the previous two commands can be rewritten like this:

```
rc whello
```

To load Windows and run the application, type

```
win whello
```

Linking from the command line

To link WHELLO.OBJ with the correct libraries and startup code, invoke TLINK with the following command-line:

```
TLINK /Tw /v /c /LC:\BORLANDC\LIB c0ws whello, whello, , import cwins cs,
whello
```

The TLINK command line is composed of options and five file names or groups of file names; each file or group of files is separated by a comma.

The **/Tw** option means to link for (target) Windows, **/v** tells TLINK to include debugging information, and **/c** forces case to be significant in public and external symbols. **/L**, followed by a path name, tells TLINK where to look for library files and for the startup .OBJ code.

The object files to link are listed next in the command line. C0WS.OBJ is the initialization module for the small memory model, and WHELLO.OBJ is program module for this application. The .OBJ extension is assumed for both these files.

The next file on the command line, WHELLO, is the name you want TLINK to give the executable file. The .EXE extension is assumed when you create a Windows application, and the .DLL extension is assumed when you create a DLL. For more details on how TLINK knows whether you want an .EXE or .DLL, see the section "Linker options" on page 116.

The next file on the command line is the name you want to give the map file. If no name is given, as in this example, TLINK gives the map file the name of the executable and adds the .MAP extension. After you run this command, you'll notice the file WHELLO.MAP in the examples directory.

The library files to link are listed after the map file. CWINS.LIB is the small memory model run-time library for Windows, CS.LIB is the regular run-time library, and IMPORT.LIB is the library that provides access to the built-in Windows functions. The .LIB extension is assumed for all library files.

The last file name on the TLINK command line is the module definition file, WHELLO.DEF (the .DEF extension is assumed). Module definition files are described briefly on page 105, and in detail in Chapter 7, page 250.

Using a makefile

Since you probably won't want to type in the full command lines for the command line compiler and TLINK every time you want to build a Windows application, it's a good idea to create a makefile for your application.

The makefile for the WHELLO application is WHELLO.MAK. Note that for this example, the libraries are in C:\BORLANDC\LIB, and the include files are in C:\BORLANDC\INCLUDE. The following section explains each rule in the makefile.

To run MAKE on this makefile, type

```
make -fwhello.mak
```

The first rule tells MAKE how to make the final executable from WHELLO.EXE and a WHELLO.RES, and how to make the intermediate executable from the object file and the module definition file. (See the alternate makefile at the end of this section for a more generalized approach to building a Windows application.)

```
whello.exe: whello.obj whello.def whello.res
   tlink /Tw /v /n /c C:\BORLANDC\LIB\c0ws whello,\
         whello,\
         ,\
         C:\BORLANDC\LIB\cwins C:\BORLANDC\LIB\cs
C:\BORLANDC\LIB\import,\
         whello
   rc whello.res
```

The next rule tells MAKE how to make required .OBJ files from .CPP files of the same name. The options are: make a Windows application (**-W**), compile only (**-c**), use the small memory model (**-ms**), and include debugging info (**-v**).

```
.cpp.obj:
    BCC -c -ms -v -W $<
```

This last rule tells MAKE how to make required .RES files (final resource files) from .RC files of the same name.

```
.rc.res:
    rc -r -iC:\BORLANDC\INCLUDE $<
```

The **–r** option tells the Resource Compiler to compile the resources only (instead of also adding them to the executable of the same name). The **–i** options specifies the directory in which to search for include files.

Another makefile for Windows

The following makefile is a more general-purpose makefile than the one shown previously. It can be easily modified by redefining the macros OBJS, INCPATH, and FLAGS. TLINK is not invoked in a separate rule; instead, BCC invokes TLINK automatically.

```
OBJS = whello.obj
INCPATH = C:\BORLANDC\INCLUDE
FLAGS = -W -v -I$(INCPATH)

test.exe: $(OBJS) whello.def whello.res
```

```
    BCC $(FLAGS) -ewhello.exe @&&!
$(OBJS)
!
    rc whello.res

.c.obj:
    BCC -c $(FLAGS) {$< }

.cpp.obj:
    BCC -c $(FLAGS) {$< }

.rc.res:
    rc -r -i$(INCPATH) $<
```

Prologs and epilogs

When you compile a module for Windows, the compiler needs to know which kind of prolog and epilog to create for each of a module's functions. Settings in the IDE and options for the command-line compiler control the creation of the prolog and epilog. The prolog and epilog perform several functions, including ensuring that the correct data segment is active during callback functions, and marking near and far stack frames for the Windows stack-crawling mechanism.

The need for prologs and epilogs is not new to Windows; they must be generated for code intended for DOS as well. However, if the program is intended for Windows, the compiler generates a different prolog and epilog than it would for DOS.

The prolog and epilog code is automatically generated by the compiler, though various compiler options or IDE options dictate the exact instructions contained in the code.

The following list describes the effects of the different Prolog/Epilog Code Generation options and their corresponding command-line compiler options. To set these options in the IDE, choose **O**ptions | **C**ompiler | Entry/Exit Code.

- Windows All Functions Exportable (**-W**). This option creates a Windows application object module with all far functions exportable.

  See page 3 for description and usage of the ***_export*** *keyword.*

  This is the most general kind of Windows application module, although not necessarily the most efficient. The compiler generates a prolog and epilog for every far function that makes the function exportable. This does not mean that all far functions actually will be exported, it only means that the function can be exported. In order to actually export one of these functions, you must either use the **_export** keyword or add an entry for the function name in the EXPORTS section of the module definition file.

- Windows Explicit Functions Exported (**-WE**). This option creates an object module with only those functions marked as **_export** exportable.

  Since, in any given application module, many of the functions will not be exported, it is not necessary for the compiler to include the special prolog and epilog for exportable functions unless a particular function is known to be exported. The **_export** keyword in a function definition tells the compiler to use the special prolog and epilog required for exported functions. All functions not flagged with **_export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

  Note that the Windows explicit functions exported option *only* works in conjunction with the **_export** keyword. This option does not export those functions listed in the EXPORTS section of a module definition file. In fact, you can't use this option and provide the names of the exported functions in the EXPORTS section. If you do, the compiler will generate prolog and epilog code that is incompatible with exported functions; incorrect behavior will result when these functions are called.

- Windows Smart Callbacks (**-WS**). This option creates an object module with functions using smart callbacks.

  This form of prolog and epilog makes use of the assumption that DS = SS; in other words, that the default data segment is the same as the stack segment; this eliminates the need for the special Windows code (called a thunk) that is created for exported functions. This form of prolog and epilog can improve performance because calls to functions in the module do not have to be redirected through the thunks.

  Exported functions here do not need the **_export** keyword, and do not need to be listed in the EXPORTS section of the module definition file, because the linker does not need to create an export entry for them in the executable.

  When you use functions compiled and linked with smart callbacks, you do not need to precede them with a call to MakeProcInstance (which rewrites the function's prolog in such a way that it uses a smart callback).

Because of the assumption that DS = SS, you can't use this option for modules in a DLL (applications are fine, but not DLLs). Furthermore, you must not explicitly change DS in your program (not a very safe practice under Windows in any circumstance).

- Windows DLL All Functions Exportable (**-WD**). This option creates a DLL object module with all functions exportable.

  There are no smart callbacks for DLLs since DLLs assume DS != SS.

  This prolog and epilog code is used for functions which will reside in a DLL. It also supports the exporting of these functions. This is similar to the corresponding non-DLL option, Object Module With All Functions Exportable.
- Windows DLL Explicit Functions Exported (**-WDE**).

  This prolog and epilog code is used for functions which will reside in a DLL. However, any functions which are to be exported must explicitly specify **_export** in the function definition. This is similar to the corresponding non-DLL option, Object module with only explicitly designated functions exportable.

The _export keyword

The keyword **_export** in a function definition is used to tell the compiler to compile the function as exportable. It also tells the linker to export the function. In a function declaration, **_export** immediately precedes the function name; for example,

```
LONG FAR PASCAL _export MainWindowProc( HWND hWnd, unsigned iMessage,
                                        WORD wParam, LONG lParam )
```

You can also use **_export** with a C++ class definition; see page 127.

Prologs, epilogs, and exports: a summary

There are two steps to exporting a function. First, the compiler must create the correct prolog and epilog for the function; if so, the function is called exportable. Second, the linker must create an entry for every export function in the header section of the executable. All of this occurs so that the correct data segment can be bound to the function at run-time.

If a function is flagged with the **_export** keyword and any of the Windows compiler options is used, it will be compiled as exportable and linked as an export.

If a function is *not* flagged with the **_export** keyword, then Borland C++ will take one of the following actions:

- If you compile with the **–W** or **–WD** option (or with the IDE equivalent of either option), the function will be compiled as exportable.

  If the function is listed in the EXPORTS section of the module definition file, then the function will be linked as an export. If it is not listed in the module definition file, or if no module definition file is linked, then it won't be linked as an export.
- If you compile with the **–WE** or **–WDE** option (or with the IDE equivalent of either option), the function will *not* be compiled as exportable. Including this function in the EXPORTS section of the module definition will cause it be exported, but, because the prolog is incorrect, the program will run incorrectly. You may get the Windows error message, "Unrecoverable application error".

The following table summarizes the effect of the combination of the Windows compiler options and the **_export** keyword:

Table 3.1: Compiler options and the **_export** keyword

Function flagged with **_export**?	Yes	Yes	Yes	Yes	No	No	No	No
Function listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
And the compiler option is:	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**
Will function be exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	Yes	Yes	Yes	Yes	Yes	Yes*	No**	No

* The function will be exported in some sense, but, because the prolog and epilog won't be correct, the function won't work as expected.

** This combination also makes little sense. It's inefficient to compile all functions as exportable if you don't actually export some of them.

Memory models

You can use the small, medium, compact, or large memory models with any kind of Windows executable, including DLLs. Do not use the tiny or the huge model for any Windows executable. Borland C++ doesn't restrict you from setting the tiny or huge model either on the command line or in the IDE, but the code that is generated won't work under Windows. See the section "Linking .OBJ and .LIB files for DLLs" on page 118 for more information.

Linking for Windows

In general, Borland C++ needs to take object files compiled with the correct Windows options and then link them with the proper Windows initialization code, run-time and math libraries, and a module definition file. Settings in the Linker dialog box in the IDE do this for you automatically; if you use TLINK, you must specify all the options and files.

Linking in the IDE

With the Linker dialog box in the IDE, you can set link options for a Windows application or DLL. Options in the IDE override settings in the module definition file. This means if you check the Windows EXE box instead of the Windows DLL box, and the module definition file has a LIBRARY statement instead of a NAME statement, the file will be linked as a Windows application, not a DLL.

The linker uses the C0W*x*.OBJ initialization file for applications and the C0D*x*.OBJ initialization file for DLLs, where *x* depends on the memory model set in the Code Generation dialog box. For both Windows options, the linker uses the current project object files and libraries, IMPORT.LIB, CWIN*x*.LIB, MATH*x*.LIB, and C*x*.LIB.

Borland C++ allows you to override the default setting for a memory model, even with an incorrect model. See the discussion of the Code Generation dialog box on page 61.

Linking with TLINK

Starting with this version, TLINK is sensitive to the case of its options.

TLINK is discussed in detail in Chapter 7, "Utilities". This section discusses only those aspects of TLINK that affect linking a Windows executable.

To provide a way to link a module definition file, the new syntax of the TLINK command line is:

TLINK *objfiles, exefile, mapfile, libfiles, deffile*

For a list of TLINK messages (errors and warnings), see Chapter 7.

Linker options

There are three options that you can pass to TLINK to control its linkage of Windows executables and DLLs.

- Use the **/Tw** option to create a Windows .EXE or .DLL according to the settings in the module definition file. If you have a NAME statement in the module definition file, TLINK will link it as a Windows executable; if you have a LIBRARY statement in the .DEF file, the files will be linked as a DLL.

  If no module definition file is specified on the TLINK command line, this option causes the files to be linked as a Windows .EXE.

You don't need this option if you are using a module definition file in which the EXETYPE statement specifies WINDOWS.

- Use the **/Twe** option to specify a Windows executable. This overrides settings in the module definition file. For instance, even if you have a LIBRARY statement in the include .DEF file, TLINK will link the files as an .EXE.
- Use the **/Twd** option to specify a Windows DLL. This overrides settings in the module definition file.

When you're linking a Windows executable, do *not* use the **/o** option to overlay files, or the **/t** or **/Tdc** option to make a .COM file.

Linking .OBJ and .LIB files

The list of object files must begin with the file C0W*x*.OBJ or C0D*x*.OBJ (for DLLs), followed by the names of the other object files to link. User libraries and IMPORT.LIB can be included anywhere on the list, although, by convention, they are usually listed before the standard libraries. The other required libraries must be in this order:

Important! *Do* ***not*** *link in EMU.LIB or FP87.LIB. Borland C++ takes care of the floating-point math automatically.*

- CWIN*x*.LIB
- MATH*x*.LIB
- C*x*.LIB

To create a Windows application executable, you might use this response file, named WINRESP:

```
/Tw /c \BORLANDC\LIB\C0WS winapp1 winapp2
winapp
winapp
\BORLANDC\LIB\IMPORT \BORLANDC\LIB\CWINS \BORLANDC\LIB\CS
winapp.def
```

where

- The **/Tw** option tells TLINK to generate a Windows application or DLL. If a module definition file were not included in the link, TLINK would create a Windows application. If the module definition file is included and it contains instructions to create a DLL, then TLINK will create a DLL.
- The **/c** option tells TLINK to be sensitive to case during linking.
- \BORLANDC\LIB\C0WS is the standard Windows initialization file and WINAPP1 and WINAPP2 are the module's object files.
- WINAPP is the name of the target Windows executable.

- TLINK will name the map file WINAPP.MAP.
- \BORLANDC\LIB\CWINS is the small memory model run-time library for Windows, LIB\CS is the regular run-time library, and \BORLANDC\LIB\IMPORT is the library that provides access to the built-in Windows functions.
- WINAPP.DEF is the Windows module definition file for the object files named.

To use this response file on the TLINK command line, type

```
TLINK @winresp
```

➡ After linking the application or DLL, you *must* invoke the Resource Compiler to add resources to the image. The Windows 3.*x* Resource Compiler also marks the image as Windows 3.*x* compatible. Even if you have no resources, you need to run the Resource Compiler.

Linking .OBJ and .LIB files for DLLs

You need to link different .OBJ and .LIB files for a DLL than for a Windows application. If the linker is invoked either from the IDE or from the command-line compiler BCC or BCCX, the correct .OBJ and .LIB files will be linked in automatically. If you invoke TLINK explicitly, then you need to know which files to link in for a DLL. The following table summarizes the memory models, startup files, and libraries:

Table 3.2
Startup and library files for DLLs

Model	**Startup file**	**Library**
Small	C0DS.OBJ	CWINC.LIB
Compact	C0DC.OBJ	CWINC.LIB
Medium	C0DM.OBJ	CWINL.LIB
Large	C0DL.OBJ	CWINL.LIB

The compact memory model library is used for both small and compact because it creates far data pointers and near code pointers. The large memory model library is used for both medium and large because it creates far data pointers as well as far code pointers. DLLs can only have far pointers to data; near pointers are not allowed.

Building a project for a Windows program

You can use the Project Manager to build Windows applications and DLLs. Building a Windows program usually requires adding a module definition file (.DEF) and resource file (.RC or .RES) to the project.

Specifying an .RC file is similar to specifying a source file in a project. The Project Manager will invoke the Resource Compiler once to compile it to an .RES file, and a second time to bind the .RES to the module and to mark the module as Windows-compatible. Specifying an .RES file is similar to specifying an object file. The Project Manager will invoke the Resource Compiler only to bind it to the module and to mark the module as Windows-compatible.

For example, if you enter HELLO.CPP, HELLO.RC, and HELLO.DEF into a project, the Borland C++ Project Manager will

- create HELLO.OBJ by compiling HELLO.CPP with the C++ compiler
- create HELLO.RES by compiling HELLO.RC with the Resource Compiler
- create HELLO.EXE by linking HELLO.OBJ with its appropriate libraries, using information contained in HELLO.DEF
- create the final HELLO.EXE by using the Resource Compiler to bind the resources contained in HELLO.RES to HELLO.EXE

When you add an .RC file to a project, the Project Manager automatically assigns the default translator to be the Resource Compiler. In addition, the default output name is *file*.RES (not *file*.OBJ). Finally, "Exclude from Link" is selected because TLINK should not link the resulting .RES file.

During a make, the Project Manager recompiles the .RC file if it is newer than the .RES file, in the same way that it recompiles HELLO.C if it is newer than HELLO.OBJ. No autodependencies are checked because that information is not available.

During a make, the Project Manager runs the Resource Compiler after any relink because the Resource Compiler also marks the image as Windows 3.*x* compatible. Even if you have no resources, you need to run the Resource Compiler.

The Project Manager will not attempt to compile a file with the .DEF extension.

WinMain

The **WinMain** function is the main entry point for a Windows application; you must supply it.

The following parameters are passed to **WinMain**:

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
```

The HANDLE and LPSTR types are defined in windows.h.

HANDLE *hInstance* is the instance handle of the application. Each instance of a Windows application will have a unique instance handle. This handle is used as an argument to several Windows functions and can be used to distinguish between multiple instances of a given application.

HANDLE *hPrevInstance* is the handle of the previous instance of this application. *hPrevInstance* is NULL if this is the first instance.

LPSTR *lpCmdLine* is a far pointer to a null-terminated command line string. This value can be specified when invoking the application from the program manager or from a call to **WinExec**.

int *nCmdShow* is an integer which specifies how to display the application's window. This value should be passed to ShowWindow. See documentation on ShowWindow for the possible values *nCmdShow* may take on.

The return value from **WinMain** is not currently used by Windows. However, it can be useful during debugging since Turbo Debugger can display this value upon termination of your program.

The Resource Compiler

The Resource Compiler does three things

- it compiles resource script files (.RC) and the resource files (.ICO, .CUR, and so on) into a binary file (.RES)
- it binds compiled resources to a compiled and linked application
- it marks the .EXE or .DLL as Windows 3.*x* compatible

Windows resources are icons, dialog boxes, fonts, menus, cursors, and bitmaps. These resources are created independently of the application or DLL. See the *Resource Toolkit* for more information on creating resources.

From the IDE, the Resource Compiler is invoked by the Project Manager when building a Windows project. Any .RC file (source file) included in a project causes Borland C++ to invoke the Resource Compiler to compile it to a .RES file. Then, after TLINK has linked the project's application or DLL, the Resource Compiler marks and binds the resources to it.

From the command line, you can compile the resource files you want to use in your Windows application with the Resource Compiler. When you're ready to build the application, you use the Resource Compiler to bind the .RES file to the .EXE or .DLL.

In a make file, add the .RES file to the list of files in the explicit rule that governs the build of the final .EXE. In that rule, also add the command to invoke the Resource Compiler with the correct .RES file. You can also add a rule to invoke the Resource Compiler on an out-of-date .RES file. See the example of a make file on page 109.

Resource Compiler syntax

This is how you invoke the Resource Compiler from the command line:

RC [*options*] *ResourceFile* [*ModuleFile*]

For example, to compile WHELLO.RC file and add it to WHELLO.EXE, you would give this command line:

```
rc whello
```

This simplest form only works if the resource file and the executable file share the same name. If WHELLO.RC was instead named WHELLORS.RC, you would type

```
rc whellors whello
```

To compile only the WHELLO.RC resource file (and not add the resulting WHELLO.RES to WHELLO.EXE), use the **–R** option, like this:

```
rc -r whello
```

You would then have a WHELLO.RES file. To add WHELLO.RES to WHELLO.EXE, type

```
rc whello.res
```

To mark a module as Windows-compatible, but not add any resources to it, simply invoke the Resource Compiler with the module name (note that the file name must have one of these extensions: .EXE, .DLL, or .DRV). For example,

```
rc whello.exe
```

The following table describes the Resource Compiler options. Note that Resource Compiler options are not case sensitive (**-e** is the same as **–E**). Also, options that take no arguments can be combined (for instance, **–kpr** is legal).

Table 3.3: Resource Compiler options

Option	What it does
–?	Lists help on Resource Compiler options (also **–H**).
–d *Symbol*	Defines *Symbol* for the preprocessor.
–e	Changes location of global memory for a DLL to above the EMS bank line
–fe *FileName*	Renames the .EXE file to *FileName*.
–fo *FileName*	Renames the .RES file to *FileName*.
–h	Lists help on Resource Compiler options (also **–?**).
–i *Path*	After searching the current directory for include files and resource files, RC searches the directory named in *Path*. The **–i** option can be repeated if you want to specify more than one search path. Also see the description for the **–x** option.
–k	Turns off load optimization for segments and resources. (Normally, the Resource Compiler preloads all data segments, nondiscardable code segments, and the entry-point code segment, even if the segments were not marked as PRELOAD in the module definition file. In addition, the Resource Compiler normally places all preloaded segments in a contiguous area in the executable file.)
–l	Informs Windows that the application will be using expanded memory, according to the LIM 3.2 specification.
–lim32	Same as **–l** option.
–m	Assigns each instance of a task to a different EMS bank, if the expanded memory under Windows is configured under EMS 4.0.
–multinst	Same as **–m** option.
–p	Makes a DLL private to one or more instances of a single application, which might result in performance gains.
–r	Compile the .RC file into a .RES file, but do not add it to an .EXE.
–t	Creates application to be run only in standard mode or 386 enhanced mode (protected mode). If the user tries to run in real mode, a message will be displayed.
–v	Display all compiler progress messages (compile verbose).
–x	Excludes searching in the directories named in the INCLUDE environment variable. Also see the description for the **–i** option.

Dynamic link libraries

A dynamic link library (DLL) is a library of functions that a Windows module can call to accomplish a task. If you've written a Windows application, then you've already used DLLs. The files KERNEL.EXE, USER.EXE, and GDI.EXE are actually DLLs, not applications (as the .EXE extension implies). The references to the API functions that you call from these modules are resolved at run time (dynamic linking), instead of at link time (static linking).

Compiling and linking a DLL within the IDE

To compile and link a DLL from within the IDE, follow these steps:

1. Create the DLL source files. Optionally, create the resource file and the module definition file.
2. Choose **P**roject | **O**pen to start a new project.
3. Choose **P**roject | **A**dd item, and add the source and resource files for the DLL.
4. If you have created a module definition file for the DLL, add it to the project. (Note that Borland C++ can link without one.)
5. Choose **O**ptions | **A**pplication | Windows dynamic link library.

   To link without a module definition file for the DLL, you must have flagged every function to be exported in the DLL with the keyword **_export**. (The **_export** keyword should immediately precede the function name.) In addition, choose **O**ptions | **C**ompiler | **E**ntry/Exit Code | Windows DLL explicit functions exportable.
6. Choose **C**ompile | **B**uild all.

Compiling and linking a DLL from the command line

To compile and link a DLL composed of the source file LIBXAMP.CPP, type

```
BCC -WD libxamp.cpp
```

The command-line compiler takes care of linking in the correct startup code and libraries. The **–WD** option tells the compiler to build a Windows DLL with all functions exportable. To compile and link with explicit functions exportable, you would use the **–WDE** option and use the **_export** keyword for export functions.

To link a DLL with the command-line linker TLINK, you might use this command line

```
TLINK /Twd /v /c /LC:\BORLANDC\LIB c0ds libxamp, libxamp, , cwinc cs
import,libxamp
```

The **/Twd** option indicates a Windows DLL, **/v** tells TLINK to include debugging information, and **/c** forces case to be significant in public and external symbols. The **/L** option specifies a library and startup file search path.

See page 118 for an explanation of the library and object files needed to link a DLL.

Module definition files

A module definition file is not strictly necessary to link either a DLL or a Windows application. See page 250 for information on default module definition file replacement settings.

There are two ways to tell the linker about export functions:

- To link with a module definition file, create an EXPORTS section in the module definition file that lists all the functions that will be used by other applications. (A utility called IMPDEF can help you do this, see page 190.)
- To link without a module definition file, you must flag every function to be exported in the DLL with the keyword **_export**. In addition, when you build or link the DLL, you must choose **O**ptions | **C**ompiler | **E**ntry/Exit Code | Windows DLL Explicit Functions Exportable.

A function must be exported from a DLL before it can be imported to another DLL or application.

Import libraries

If a Windows application module or another DLL uses functions from a DLL, you have two ways to tell the linker about them:

- You can add an IMPORTS section to the module definition file and list every function from DLLs that the module will use.
- Or you can include the import library for the DLLs when you link the module. (A utility called IMPLIB creates an import library for one or more DLLs; see page 192 for details.)

Creating DLLs

The following sections provide information on the specifics of writing a DLL.

LibMain and WEP

The **LibMain** function is the main entry point for a Windows DLL; you must supply it yourself.

Windows calls **LibMain** once, when the library is first loaded. **LibMain** performs initialization for the DLL. This initialization depends almost entirely on the function of the particular DLL, but might include the following tasks:

- Unlocking the data segment with **UnlockData**, if it has been declared as MOVEABLE
- Setting up global variables for the DLL, if it uses any
- Registering Windows for the DLL

Note The DLL startup code C0Dx.OBJ initializes the local heap automatically; you do not need to include code in **LibMain** to do this.

This function is called by Windows (actually by the run-time library) upon DLL initialization. You must supply it when building a DLL. The following parameters are passed to **LibMain**:

```
int PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,
                    WORD cbHeapSize, LPSTR lpCmdLine)
```

HANDLE, WORD, and LPSTR are defined in windows.h

HANDLE *hInstance* is the instance handle of the DLL.

WORD *wDataSeg* is the value of the data segment (DS) register.

WORD *cbHeapSize* is the size of the local heap specified in the module definition file for the DLL.

LPSTR *lpCmdLine* is a far pointer to the command line specified when the DLL was loaded. This is almost always null since DLLs are typically loaded automatically with no parameters. It is possible, however, to supply a command line to a DLL when it is loaded explicitly.

The return value for **LibMain** is either 1 (successful initialization) or 0 (failure in inititalization). If 0, Windows will unload the DLL from memory.

The exit point of a DLL is the function **WEP** (which stands for Windows exit procedure). This function is not necessary in a DLL (since the Borland C++ run-time libraries provide a default one) but can be supplied by the writer of a DLL to perform any cleanup of the DLL before it is unloaded from memory. Windows will call the **WEP** procedure just prior to unloading the DLL.

WEP should perform any cleanup required for the DLL. Under Borland C++, **WEP** does not need to be exported. Borland C++ defines its own **WEP** that calls your **WEP**, and then performs system cleanup. This is the prototype for **WEP**:

```
int FAR PASCAL WEP (int nParameter)
```

int *nParameter* is either WEP_SYSTEMEXIT or WEP_FREE_DLL. The former means that all of Windows is shutting down and the latter indicates that just this DLL is being unloaded.

WEP should return 1 to indicate success. Windows currently doesn't do anything with this return value.

Pointers and memory

Functions in a DLL are not linked directly into a Windows application, they are called at run time. This means that calls to DLL functions will be far calls, because the DLL will have a different code segment than the application. The data used by called DLL functions will need to be far as well.

Let's say you have a Windows application called APP1, a DLL defined by LSOURCE1.C, and a header file for that DLL called lsource1.h. Function **f1**, which operates on a string, is called by the application.

If you want the function to work correctly regardless of the memory model the DLL will be compiled under, you need to explicitly make the function and its data far. In the header file, the function prototype would take this form:

```
extern FAR f(char FAR *dstring);
```

In the DLL, the implementation of the function would take this form:

```
far f1(char far *dstring)
{
   ...
}
```

For the function to be used by the application, the function would also need to be compiled as exportable and then exported. To accomplish this, you can either compile the DLL with all functions exportable (**-WD**) and list f1 in the EXPORTS section of the module definition file, or you can flag the function with the _export keyword, like so:

```
far _export f1(char far *dstring)
{
   ...
}
```

If you compile the DLL under the large model (far data, far code), then you don't need to explicitly define the function or its data far in the DLL. In the header file, the prototype would still take this form

```
extern FAR f(char FAR *dstring);
```

because the prototype would need to be correct for a module compiled with a smaller memory model. But in the DLL, the function could be defined like this:

```
_export f1(char *dstring)
{
   ...
}
```

Static data in DLLs

Through a DLL's functions, all applications using the DLL have access to that DLL's global data. A particular function will use the same data, regardless of the application that called it. If you want a DLL's global data to be protected for use by a single application, you would need to write that protection yourself. The DLL itself does not have a mechanism for making global data available to a single application. If you need data to be private for a given caller of a DLL, you will need to dynamically allocate the data and manage the access to that data manually. Static data in a DLL is global to all callers of a DLL.

C++ classes and pointers

A C++ class that is used only inside a DLL does not need to be declared as far. If the class is to be used from another DLL or a Windows application, then it requires special handling.

All of the the data members and member functions of a shared class need to be far. This can be accomplished by declaring the class members as **far** or compiling the DLL under the large memory model. The classes also must be exported, which can be accomplished two different ways:

- either include the names of all the class members in the EXPORTS section of the module definition file, then compile the DLL with the compiler option that makes all functions exportable (**-WD**)
- or mark the entire class with the **_export** keyword, and compile the DLL with the compiler option that makes explicit functions exportable (**-WDE**)

C++ classes use virtual table pointers and include a hidden **this** pointer. Both of these pointers need to be far pointers as well. There are two basic ways to accomplish this.

One way is to simply compile the DLL modules and the application using the DLL with the Far C++ Virtual Table Pointers option (**O**ptions | **C**ompiler | **C**++ options in the IDE or **–Vf** from the command line). This causes all virtual table pointers and **this** parameters to be full 32-bit pointers. The advantage of this approach is that it does not require any source code changes. It may be less efficient than possible, though; all classes, shared or not, suffer the overhead of 32-bit pointers.

A more efficient approach is to declare the shared classes **huge** instead of **far**. This tells the compiler to use full 32-bit pointers for those classes only. Note that a huge class can only inherit from other huge classes. Here is an example of a huge class declaration:

```
class   huge   DLLclass
{
   ... etc ...
};
```

For a class that is defined in a DLL to be usable from a Windows application, its non-inline member functions and static data members must be made available by making them exported names. This can be done by adding their public (mangled) names to the EXPORTS section of the DLL module definition file, but this can be rather tedious.

There's an easier alternative: declare the classes to be exported as **_export**. Whenever a class is declared as **_export**, Borland C++ treats it as huge (with 32-bit pointers), and automatically exports all of its non-inline member functions and static data members. If you declare a class as **_export**, you can't also declare it as **far** or **huge**. (**_export** implies **huge**, which implies **far**.)

If you declare the class in an include file that is included both by the DLL source files and by the source files of the application using the DLL, such a class should be declared **_export** when compiling the DLL, and merely **huge** when compiling the application. To do this, you can use the _ _DLL_ _ macro, which is defined by the compiler when it's building a DLL. The following code could be a part of an include file that defines a shared class:

```
#ifdef   __DLL__
#define  EXPORT   _export
#else
#define  EXPORT   huge
#endif

class  EXPORT  DLLclass
{
   ... etc ...
};
```

Note that the compiler encodes (in the mangled name) the information that a given class member is a member of a huge class. This ensures that when a program using huge and non-huge classes is linked, any mismatches are caught by the linker.

C H A P T E R

4

Managing multi-file projects

Since most programs consist of more than one file, having a way to automatically identify those that need to be recompiled and linked would be ideal. Borland C++'s built-in Project Manager does just that and more.

The Project Manager allows you to specify the files belonging to the project. Whenever you rebuild your project, the Project Manager automatically updates the information kept in the project file. This project file, which includes

- all the files in the project
- where to find them on the disk
- which files depend on which other files being compiled first (auto-dependency issues)
- which compilers and command-line options need to be used when creating each part of the program
- where to put the resulting program
- code size, data size, and number of lines from the last compile

Using the Project Manager is easy. To build a project,

- pick a name for the project file (from **P**roject | **O**pen Project)
- add source files using the **P**roject | **A**dd Item dialog box
- tell Borland C++ to **C**ompile | **M**ake EXE File

Then, with the project-management commands available on the **P**roject menu, you can

- add or delete files from your project
- set options for a file in the project
- view included files for a specific file in the project

All the files in this chapter are in the Examples directory.

Let's take a look at an example of how the Project Manager works.

Using the project manager

If you have project files from Turbo C, use the utility PRJCNVT.EXE to convert them; for details, see UTIL.DOC on disk.

Suppose you have a program that consists of a main source file, MYMAIN.C, a support file, MYFUNCS.C, that contains functions and data referenced from the main file, and myfuncs.h. MYMAIN.C looks like this:

```
#include <stdio.h>
#include "myfuncs.h"

main (int argc, char *argv[])
{
   char *s;

   if (argc > 1)
      s = argv[1];
   else
      s = "the universe";

   printf("%s %s.\n",GetString(),s);
}
```

MYFUNCS.C looks like this:

```
char ss[] = "The restaurant at the end of";

char *GetString(void)
{
   return ss;
}
```

And myfuncs.h looks like this:

```
extern char *GetString(void);
```

These files make up the program that we'll now describe to the Project Manager.

These names can be the same (except for the extensions), but they don't have to be. The name of your executable file (and any map file produced by the linker) is based on the project file's name.

The first step is to tell Borland C++ the name of the project file that you're going to use: Call it MYPROG.PRJ. Notice that the name of the project file is not the same as the name of the main file (MYMAIN.C). And in this case, the executable file will be MYPROG.EXE (and if you choose to generate it, the map file will be MYPROG.MAP).

Press *Alt-P* to go to the **P**roject menu and choose **O**pen Project. This brings up the Load Project File dialog box, which contains a list of all the files in the current directory with the extension .PRJ, and date, time, and size information about the first project file. Since you're starting a new file, type in the name `MYPROG` in the Load Project File input box.

Notice that once a project is opened, the **A**dd Item, **D**elete Item, **L**ocal Options, and **I**nclude Files options are enabled on the **P**roject menu.

If the project file you load is in another directory, the current directory will be set to where the project file is loaded.

You can keep your project file in any directory; to put it somewhere other than the current directory, just specify the path as part of the file name. (You must also specify the path for source files if they're in different directories.) Note that all files and corresponding paths are relative to the directory where the project file is loaded from. After you enter the project file name, you'll see a Project window.

The Project window contains the current project file name (MYPROG) and information about the files you've selected to be part of your current project. For each file, the name and path (location) are shown. Once the file is compiled, it also shows the number of lines in the file and the amount of code and data in bytes generated by the compiler.

The status line at the bottom of the screen shows which actions can be performed at this point: *F1* gives you help, *Ins* adds files to the Project, *Del* deletes a file from the Project, *Ctrl-O* lets you select options for a file, *Spacebar* lets you view information about the include files required by a file in the Project, *Enter* opens an editor window for the currently selected file, and *F10* takes you to the main menu. Press *Ins* now to add a file to the project list.

*You can change the filename specification to whatever you want with the Name input box; *.C is the default.*

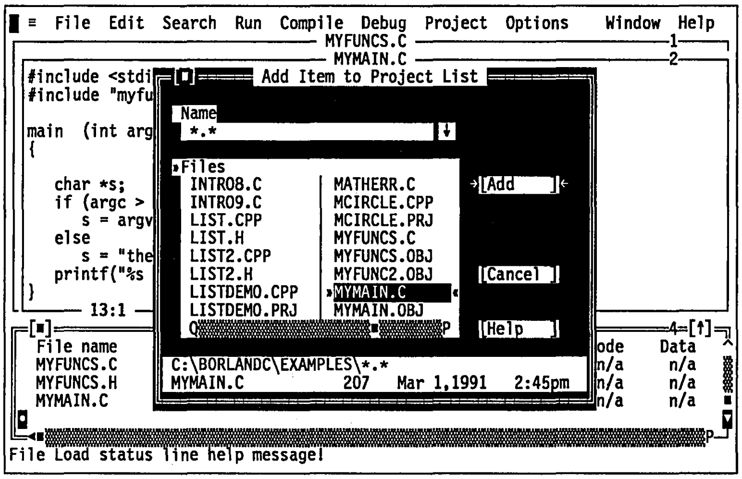
The Add Item to Project List dialog box appears; this dialog lets you select and add source files to your project. The Files list box shows all files with the .C extension in the current directory. (MYMAIN.C and MYFUNCS.C both appear in this list.) Three action buttons are available: Add, Cancel, and Help.

If you copy the wrong file to the Project window, press Esc to return to the Project window, then Del to remove the current file (marked with a •) from the list.

Since the Add button is the default, you can place a file in the Project window by typing its name in the Name input box and pressing *Enter* or by choosing it in the Files list box. You can also search for a file in the Files list box by typing the first few letters of the one you want. In this case, typing `my` should take you right to MYFUNCS.C; press *Enter.* You'll see that MYFUNCS gets added to the Project window and then you're returned to the Add Item dialog box to add another file. Go ahead and add MYMAIN.C. Borland C++ will compile files in the exact order they appear in the project.

Note that the Add button commits your change; pressing Esc when you're in the dialog box just puts the dialog box away.

Press *Esc* to close the dialog box and return to the Project window. Notice that the Lines, Code, and Data fields in the Project window show `n/a`. This means the information is not available until the modules are actually compiled.


≡ File Edit Search Run Compile Debug Project Options Window Help
Add Item to Project List
Name
Files
Add
Cancel
Help

```
MYFUNCS.C                                   1
MYMAIN.C                                    2
#include <stdi
#include "myfu
main  (int arg       *.*
{
   char *s;          INTRO8.C          MATHERR.C
   if (argc >        INTRO9.C          MCIRCLE.CPP
      s = argv       LIST.CPP          MCIRCLE.PRJ
   else              LIST.H            MYFUNCS.C
      s = "the       LIST2.CPP         MYFUNCS.OBJ
   printf("%s        LIST2.H           MYFUNC2.OBJ
}                    LISTDEMO.CPP     »MYMAIN.C
      13:1           LISTDEMO.PRJ      MYMAIN.OBJ
                                                            4
 File name                                          ode    Data
 MYFUNCS.C        C:\BORLANDC\EXAMPLES\*.*           n/a    n/a
 MYFUNCS.H        MYMAIN.C       207   Mar 1,1991   2:45pm   n/a    n/a
 MYMAIN.C                                           n/a    n/a
File Load status line help message!
```

After all compiler options and directories have been set, Borland C++ will know everything it needs to know about how to build the program called MYPROG.EXE using the source code in MYMAIN.C, MYFUNCS.C, and myfuncs.h. Now you'll actually build the project.

You can also view your output by choosing Window | Output.

Press *F10* to go to the main menu. Now make MYPROG.EXE by pressing *F9* (or choose Compile | **M**ake EXE File). Then run your program by pressing *Ctrl-F9* (or choose **R**un | **R**un). To view your output, choose **W**indow | **U**ser Screen (or press *Alt-F5*), then press any key to return to the IDE.

When you leave the IDE, the project file you've been working on is automatically saved on disk; you can disable this by unchecking Project in the Preferences dialog box (**O**ptions | **E**nvironment).

For more information on .PRJ and .DSK files, refer to the section, "Project and configuration files," in Chapter 2.

The saved project consists of two files: the project file (.PRJ) and the desktop file (.DSK). The project file contains the information required to build the project's related executable (.EXE). The build information consists of compiler options, INCLUDE/LIB/OUTPUT paths, linker options, make options, and transfer items. The desktop file contains the state of all windows at the last time you were using the project.

You can specify a project to load on the DOS command line like this: `BC myprog.prj`.

The next time you use Borland C++, you can jump right into your project by reloading the project file. Borland C++ automatically loads a project file if it is the only .PRJ file in the current directory; otherwise the default project and desktop (TCDEF.*) are loaded. Since your program files and their corresponding paths are relative to the project file's directory, you can work on any project by moving to the project file's directory and bringing up Borland C++. The correct file will be loaded for you automatically. If no project file is found in the current directory, the default project file is loaded.

Error tracking

As with single-file programs, syntax errors that generate compiler warning and error messages in multifile programs can be selected and viewed from the Message window.

To see this, let's introduce some syntax errors into the two files, MYMAIN.C and MYFUNCS.C. From MYMAIN.C, remove the first angle bracket in the first line and remove the *c* in **char** from the fifth line. These changes will generate five errors and two warnings in MYMAIN.

In MYFUNCS.C, remove the first *r* from `return` in the fifth line. This change will produce two errors and one warning.

Changing these files makes them out of date with their object files, so doing a make will recompile them.

Since you want to see the effect of tracking in multiple files, you need to modify the criterion Borland C++ uses to decide when to stop the make process. This is done by setting a radio button in the Make dialog box (**O**ptions | **M**ake).

Stopping a make

You can choose the type of message you want the make to stop on by setting one of the Break Make On options in the Make dialog box (**O**ptions | **M**ake). The default is Errors, which is normally the setting you'd want to use. However, you can have a make stop after compiling a file with warnings, with errors, or with fatal errors, or have it stop before it tries to link.

The usefulness of each of these modes is really determined by the way you like to fix errors and warnings. If you like to fix errors and warnings as soon as you see them, you should set Break Make On to Warnings or maybe to Errors. If you prefer to get an entire list of errors in all the source files before fixing them up, you should set the radio button to Fatal Errors or to Link. To demonstrate errors in multiple files, choose Fatal Errors in the Make dialog box.

Syntax errors in multiple source files

Since you've already introduced syntax errors into MYMAIN.C and MYFUNCS.C, go ahead and press *F9* (Make) to "make the project." The Compiling window shows the files being compiled and the number of errors and warnings in each file and the total for the make. Press any key when the `Errors: Press any key` message flashes.

Your cursor is now positioned on the first error or warning in the Message window. If the file that the message refers to is in the editor, the highlight bar in the Edit window shows you where the compiler detected a problem. You can scroll up and down in the Message window to view the different messages.

Note that there is a "Compiling" message for each source file that was compiled. These messages serve as file boundaries, separating the various messages generated by each module and its include files. When you scroll to a message generated in a different source file, the Edit window will only track in files that are currently loaded.

Thus, moving to a message that refers to a file that is not loaded causes the Edit window's highlight bar to turn off. Press *Spacebar* to load that file *and* continue tracking; the highlight bar will reappear. If you choose one of these messages (that is, press *Enter* when positioned on it), Borland C++ loads the file it references into an Edit window and places the cursor on the error. If you then return to the Message window (press *Alt-W M*), tracking resumes in that file.

The Source Tracking options in the Preferences dialog box (**O**ptions | **E**nvironment) help you determine which window a file is loaded into. You can use these settings when you're message tracking and debug stepping.

Note that **P**revious message and **N**ext message (*Alt-F7* and *Alt-F8*) are affected by the Source Tracking setting. These commands will always find the next or previous error and will load the file using the method specified by the Source Tracking setting.

Saving or deleting messages

Normally, whenever you start to make a project, the Message window is cleared out to make room for new messages. Sometimes, however, it is desirable to keep messages around between makes.

Consider the following example: You have a project that has many source files and your program is set to stop on Errors. In this case, after compiling many files with warnings, one error in one file stops the make. You fix that error and want to find out if the compiler will accept the fix. But if you do a make or compile again, you lose your earlier warning messages. To avoid this, check Save Old Messages in the Preferences dialog box (**O**ptions | **E**nvironment). This way the only messages removed are the ones that result from the files you *re*compile. Thus, the old messages for a given file are replaced with any new messages that the compiler may generate.

You can always get rid of all your messages by choosing **C**ompile | **R**emove Messages, which zaps all the current messages. Unchecking Save Old Messages and running another make will also get rid of any old messages.

The power of the Project Manager

When you made your previous project, you dealt with the most basic situation: a list of C source file names. The Project Manager provides you with a lot of power to go beyond this simple situation.

Autodependency checking

The Project Manager collects autodependency information at compile time and caches these so that only files compiled outside the IDE need to be processed. The Project Manager can automatically check dependencies between source files in the project list (including files they themselves include) and their corresponding object files. This is useful when a particular C source file depends on other files. It is common for a C source to include several header files (.h files) that define the interface to external routines. If the interface to those routines changes, you'll want the file that uses those routines to be recompiled.

If you've checked the Auto-Dependencies option (**Options | Make**), Make obtains time-date stamps for all .C files and the files included by these. Then make compares the date/time information of all these files with their date/time at last compile. If any date/time is different, the source file is recompiled.

If the Auto-Dependencies option is unchecked, the .C files are checked against .OBJ files. If earlier .C files exist, the source file is recompiled.

When a file is compiled, the IDE's compiler and the command-line compiler put dependency information into the .OBJ files. The Project Manager uses this to verify that every file that was used to build the .OBJ file is checked for time and date against the time and date information in the .OBJ file. The .C source file is recompiled if the dates are different.

That's all there is to dependencies. You get the power of more traditional makes while avoiding long dependency lists.

Using different file translators

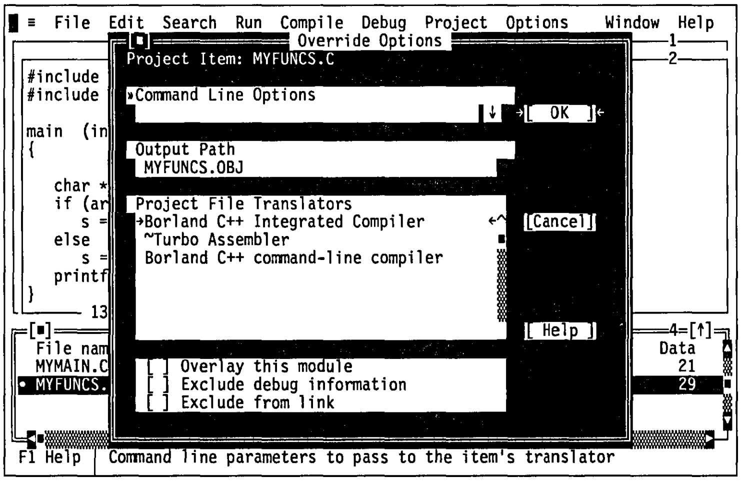
So far you've built projects that use Borland C++ as the only language translator. Many projects consist of both C code and assembler code, and possibly code written in other languages. It would be nice to have some way to tell Borland C++ how to build such modules using the same dependency checks that we've just described. With the Project Manager, you don't need to worry about forgetting to rebuild those files when you change some of the source code, or about whether you've put them in the right directory, and so on.

For every source file that you have included in the list in the Project window, you can specify

- which program (Borland C++, TASM, and so on) is to be used to make its target file
- which command-line options to give that program
- whether the module is to be an overlay
- what the resulting module is called and where it will be placed (this information is used by the project manager to locate files needed for linking)
- whether the module should contain debug information
- whether the module should be included in the link

By default, the IDE's compiler is chosen as the translator for each module, using no command-line override options, using the Output directory for output, assuming that the module is not an overlay, and assuming that debug information is not to be excluded.

Let's look at a simple example. Go to the Project window and move to the file MYFUNCS.C. Now press *Ctrl-O* to bring up the Override Options dialog box for this file:


≡ File Edit Search Run Compile Debug Project Options Window Help
Override Options
[OK]
[Cancel]
[Help]
F1 Help

```
Project Item: MYFUNCS.C

Command Line Options

Output Path
 MYFUNCS.OBJ

Project File Translators
 Borland C++ Integrated Compiler
 Turbo Assembler
 Borland C++ command-line compiler

[ ] Overlay this module
[ ] Exclude debug information
[ ] Exclude from link

Command line parameters to pass to the item's translator
```

Except for Borland C++, each of the names in the Project File Translators list box is a reference to a program defined in the Transfer dialog box (**O**ptions | **T**ransfer).

Press *Esc*, then *F10* to return to the main menu, then choose **O**ptions | **T**ransfer. The Transfer dialog box that appears contains a list of all the transfer programs currently defined. Use the arrow keys to select Turbo Assembler and press *Enter*. (Since the Edit button is the default, pressing *Enter* brings up the Modify/New Transfer Item dialog box.) Here you see that Turbo Assembler is defined as the program TASM in the current path. Notice that the Translator check box is marked with an X; this translator item is then displayed in the Override Options dialog box. Press *Esc* to return to the Transfer dialog box.

Suppose you want to compile the MYFUNCS module using the Borland C++ command-line compiler instead of the IDE's compiler. To do so, you would perform the following steps:

1. First, you need to define BCC as one of the Project File Translators in the Transfer dialog box. Cursor past the last entry in the Program Titles list, then press *Enter* to bring up the Modify/New Transfer Item dialog box. In the Program Title input box, type `Borland C++ Command-Line Compiler`; in the Program Path input box, type `BCC`; and in the command line, type `$EDNAME`.
2. Then check Translator by pressing *Spacebar* and press *Enter* (New is the default action button). Back at the Transfer dialog box, you see that Borland C++ command-line (*compiler* doesn't show) is now in the Program Titles list box. Tab to OK and press *Enter*.
3. Back in the Project window, press *Ctrl-O* to go to the Override Options dialog box again. Notice that *Borland C++ Command-Line Compiler* is now a choice on the Project File Translators list for MYFUNCS.C (as well as for all of your other files).

   Tab to the Project File Translators list box and highlight *Borland C++ Command-Line Compiler* (at this point, pressing *Enter* or tabbing to another group will choose this entry). Use the Command-Line Options input box to add any command-line options you want to give BCC when compiling MYFUNCS.

MYFUNCS.C now compiles using BCC.EXE, while all of your other source modules compile with BC.EXE. The Project Manager will apply the same criteria to MYFUNCS.C when deciding whether to recompile the module during a make as it will to all the modules that are compiled with BC.EXE.

Overriding libraries

In some cases, it's necessary to override the standard startup files or libraries. You override the startup file by placing a file called C0*x*.OBJ as the *first* name in your project file, where *x* stands for any DOS name (for example, C0MINE.OBJ). It's critical that the name start with *C0*, that it is the first file in your project, and that it have an explicit .OBJ extension.

To override the standard library, all you need to do is place a special library name anywhere in the list of names in the Project window. The name of the library must start with a *C*, followed by a letter representing the model (such as *S* for the small model); the remaining characters, up to six, can be anything you want for a file name. You must use an explicit .LIB extension (for example, CSMYFILE.LIB or CSNEW.LIB).

When the standard library is overridden, MAKE will not try to link in the math libraries (based on the Floating Point setting in the Advanced Code Generation dialog box of the **O**ptions | **C**ompiler menu). If you want these libraries linked in when you override the standard library, you must explicitly include them in the Project.

More Project Manager features

Let's take a look at some of the other features the Project Manager has to offer. When you're working on a project that involves many source files, you want to be able to easily view portions of those files, and be able to record notes about what you're doing as you're working. You'll also want to be able to quickly access files that are included by others. The Project Manager provides these features and more.

For example, expand MYMAIN.C to include a call to a function named **GetMyTime**:

```
#include <stdio.h>
#include "myfuncs.h"
```

```
#include "mytime.h"

main (int argc, char *argv[])
{
   char *s;

   if (argc > 1)
      s = argv[1];
   else
      s = "the universe";

   printf("%s %s opens at %d.\n",GetString(),s,GetMyTime(HOUR));
}
```

This code adds two include files to MYMAIN: myfuncs.h and mytime.h. These files contain the prototypes that define the **GetString** and **GetMyTime** functions, which are called from MYMAIN. myfuncs.h contains

```
extern char *GetString(void);
```

mytime.h contains

```
#define HOUR 1
#define MINUTE 2
#define SECOND 3
extern int GetMyTime(int);
```

Go ahead and put the actual code for **GetMyTime** into a new source file called MYTIME.C:

```
#include <time.h>
#include "mytime.h"

int GetMyTime(int which)
{
   struct tm    *timeptr;
   time_t       secsnow;

   time(&secsnow);
   timeptr = localtime(&secsnow);
   switch (which) {
      case HOUR:
         return (timeptr -> tm_hour);
      case MINUTE:
         return (timeptr -> tm_min);
      case SECOND:
         return (timeptr -> tm_sec);
  }
}
```

MYTIME includes the standard header file time.h, which contains the prototype of the **time** and **localtime** functions, and the definition of *tm* and *time_t*, among other things. It also includes mytime.h in order to define HOUR, MINUTE, and SECOND.

Create these new files, then use **P**roject | **O**pen Project to open MYPROG.PRJ. The files MYMAIN.C and MYFUNCS.C are still in the Project window. Now to build your expanded project, you add the file name MYTIME.C to the Project window. Press *Ins* (or choose **P**roject | **A**dd Item) to bring up the Add Item dialog box. If you placed MYTIME.C in the current directory, use the Files list box to choose it now. If MYTIME.C is in a different directory, tab to the Name input box and type in MYTIME.C and its path. Once you've used either of these methods, press *Enter* to actually add the file. The Add button is the default action button.

Now press *F9* to make the project. MYMAIN.C will be recompiled because you've made changes to it since you last compiled it. MYFUNCS.C won't be recompiled, because you haven't made any changes to it since the make in the earlier example. MYTIME.C will be compiled for the first time.

In the MYPROG project window, move to MYMAIN.C, and press *Spacebar* (or **P**roject | **I**nclude Files) to display the Include Files dialog box. This dialog box contains the name of the selected file, several buttons, and a list of include files and locations (paths). The first file in the Include Files list box is highlighted; the list box lists all the files that were included by the file MYMAIN.C. If any of the include files is located outside of the current directory, the path to the file is shown in the Location field of the list box.

As each source file is compiled, the information about which include files are included by which source files is stored in the source file's .OBJ file. If you access the Include Files dialog box before you perform a make, it might contain no files or it might have files left over from a previous compile (which may be out of date). To load one of the include files into an Edit window, highlight the file you want and press *Enter* (or click on View).

Looking at files in a project

Let's take a look at MYMAIN.C, one of the files in the Project. Simply choose the file using the arrow keys or the mouse, then press *Enter*. This brings up an edit window with MYMAIN.C loaded. Now you can make changes to the file, scroll through it, search for text, or whatever else you need to do. When you are finished with the file, save your changes if any (*F2*), then press *Alt-F3* to close the Edit window.

Suppose that after browsing around in MYMAIN.C, you realize that what you really wanted to do was look at mytime.h, one of the files that MYMAIN.C includes. Highlight MYMAIN.C in the Project window, then press *Spacebar* to bring up the Include Files dialog box for MYMAIN. (Alternatively, while MYMAIN.C is the active Edit window, select **P**roject | **I**nclude Items or *Alt-P I*.) Now choose mytime.h in the Include Files box and press the View button. This brings up an Edit window with mytime.h loaded. When you're done, press *Alt-F3* to close the mytime.h Edit window.

Notes for your project

Now that you've had a chance to see the code in MYMAIN.C and mytime.h, you've decided you'll optimize it as soon as you can. Choose **W**indow | **P**roject Notes. This brings up a new Edit window that is kept as part of your project file. Type in the following:

```
Change History:

Chuck G.

  Added check for out of memory in DBADDFIELD.

Harry B.

  Fixed bug 0183.
```

Each project maintains its own notes file, so that you can keep notes that go with the project you're currently working on; they're at the touch of a button as soon as you select the project file. Press *Alt-F3* now to close the Project Notes Edit window.

C H A P T E R

5

The editor from A to Z

You might want to read this chapter even if you are familiar with the editor in other Borland language products. The Programmer's Platform (the IDE) includes improvements to the editor. Context-sensitive help is always just a keystroke away (F1).

This chapter is a reference to the IDE's full range of editing commands. Table 5.1 contains a list of all of the editor commands; the tables and text that follow it cover those aspects of the editor that need further explanation.

Remember, this chapter is concerned *just* with the editor. For an in-depth discussion of the Programmer's Platform as a whole, refer to Chapter 1. For a detailed reference to each menu and option, see Chpater 2.

The new and the old

You can still use Borland's familiar hot key combinations to move around your file, insert, copy, and delete text, and search and replace. However, you now have two brand-new menus on the menu bar: the Edit menu and the Search menu. In addition, you can use a mouse to accomplish many of the cursor movement and block-marking key commands.

The Edit menu contains commands for cutting, copying, and pasting in a file, copying examples from Help to an Edit Window, and viewing the Clipboard. When you first start the IDE, an Edit window is already active. To open other Edit windows, go to the **F**ile menu and choose **O**pen. From an Edit window, you still press *F10* to get to the menu bar; to return to the Edit window, keep pressing *Esc* until you are out of the menus. If you have a mouse, you can also just click anywhere in the Edit window.

As always, you enter text pretty much as if you were using a typewriter. To end a line, press *Enter.* When you've entered enough lines to fill the screen, the top line scrolls off the screen. Don't worry, it isn't lost, and you can move back and forth in your text with the scrolling commands that are described later.

If you want to undo some changes, you can use the Undo command, described in detail on page 34 in Chapter 2.

Editor reference

Table 5.1 summarizes all editor commands.

The editor is much more powerful than the quick tutorial can show. In addition to the menu choices, it uses approximately 50 commands to move the cursor around, page through text, find and replace strings, and so on. These commands can be grouped into four main categories:

- Cursor movement
- Insert and delete operations
- Block operations
- Miscellaneous editing operations

Most of these commands need no explanation. Those that do are described in the text following Table 5.1.

Table 5.1
Full summary of editor commands

*A word is defined as a sequence of characters separated by one of the following: space < > , ; . ( ) ( ) ^ ` * + – / $ # _ = | ~ ? ! " % & : @ \, and all control and graphic characters.*

Movement	Command
Cursor movement commands	
Basic cursor movement	
Character left	←
Character right	→
Word left	*Ctrl* ←
Word right	*Ctrl* →
Line up	↑
Line down	↓
Scroll up one line	*Ctrl-W*
Scroll down one line	*Ctrl-Z*
Page up	*PgUp*
Page down	*PgDn*

Many of the commands in this table can also be performed with the mouse. See Chapter 2.

Table 5.1: Full summary of editor commands (continued)

Movement	Command
Long distance	
Beginning of line	*Home*
End of line	*End*
Top of window	*Ctrl Home*
Bottom of window	*Ctrl End*
Beginning of file	*Ctrl PgUp*
End of file	*Ctrl PgDn*
Beginning of block	*Ctrl-Q B*
End of block	*Ctrl-Q K*
Last cursor position	*Ctrl-Q P*
Insert and delete commands	
Insert mode on/off	**O**ptions | **E**nvironment | **E**ditor or *Ins*
Delete character left of cursor	*Backspace*
Delete character at cursor	*Del*
Delete word right	*Ctrl-T*
Insert line	*Ctrl-N*
Delete line	*Ctrl-Y*
Delete to end of line	*Ctrl-Q Y*
Block commands	
Mark block	*Shift* ↓, ↑, →, ←, *Ctrl-K B, Ctrl-K K*
Mark single word	*Ctrl-K T*
Copy block	**E**dit | **C**opy, **E**dit | **P**aste or *Ctrl-Ins, Shift-Ins*
Move block	**E**dit | Cu**t**, **E**dit | **P**aste or *Shift-Del, Shift-Ins*
Delete block	**E**dit | **C**lear or *Ctrl-Del*
Read block from disk	*Ctrl-K R*
Write block to disk	*Ctrl-K W*
Hide/display block	*Ctrl-K H*
Print block	**F**ile | **P**rint or *Ctrl-K P*
Indent block	*Ctrl-K I*
Unindent block	*Ctrl-K U*
Other editing commands	
Autoindent on/off	**O**ptions | **E**nvironment | **E**ditor*
Control character prefix**	*Ctrl-P*
Find place marker	*Ctrl-Q n****
Go to menu bar	*F10*
New file	**F**ile | **N**ew
Open file	**F**ile | **O**pen (*F3*)
Optimal fill mode on/off	**O**ptions | **E**nvironment | **E**ditor*
Pair matching	*Ctrl-Q [* and *Ctrl-Q]*
Print file	**F**ile | **P**rint
Quit IDE	**F**ile | **Q**uit (*Alt-X*)
Repeat last search	**S**earch | **S**earch Again or *Ctrl-L*
Restore error message	*Ctrl-Q W*

Table 5.1: Full summary of editor commands (continued)

Movement	Command
Undo changes	**Edit | Undo**
Return to editor from menus	*Esc*
Save	**File | Save** (*F2*)
Search	**Search | Find** or *Ctrl-Q F*
Search and replace	**Search | Replace** or *Ctrl-Q A*
Set place marker	*Ctrl-K n****
Tab	*Tab*
Tab mode	**Options | Environment | Editor***
Unindent mode	**Options | Environment | Editor***

*This command opens the Environment Options dialog box, in which you can set the appropriate check box or radio buttons.

**Enter control characters by first pressing *Ctrl-P*, then pressing the desired control character. Depending on your screen setup, control characters appear as low-intensity or inverse capital letters.

****n* represents a number from 0 to 9.

Jumping around

There are three cursor movement commands that need further explanation:

Ctrl-Q B and *Ctrl-Q K* move the cursor to the block-begin or block-end marker. Both these commands work even if the block is not displayed (see "Hide/display block" in Table 5.2). *Ctrl-Q B* works even if the block-*end* marker is not set, and *Ctrl-Q K* works even if the block-*begin* marker is not set.

Beginning of block	*Ctrl-Q B*
End of block	*Ctrl-Q K*
Last cursor position	*Ctrl-Q P*

Ctrl-Q P moves to the last position of the cursor before the last command. This command is particularly useful after a search or search-and-replace operation has been executed, and you'd like to return to where you were at before you ran the search.

Block commands

A block of text is any amount of text, from a single character to hundreds of lines, that has been surrounded with special block-marker characters. There can be only one block in a window at a time. A block is marked by placing a block-begin marker on the first character and a block-end marker after the last character of the desired portion of the text. Once marked, the block can be copied, moved, deleted, printed, or written to a file.

Table 5.2: Block commands in depth

Movement	Command(s)	Function
Mark block	*Shift* ↓, ↑, →, ←	Marks (highlights) a block as the cursor is moved. Marked text is displayed in a different intensity.
Mark single	*Ctrl-K T*	Marks a single word as a block. If the cursor is placed within a word, that word will be marked. If it is not within a word, then the word to the left of the cursor will be marked.
Copy block	Edit | **C**opy, *Ctrl-Ins* Edit | **P**aste, *Shift-Ins*	Copies a previously marked block to the Clipboard and pastes it to the current cursor position. The original block is unchanged, and the block markers are placed around the new copy of the block. If no block is marked or the cursor is within the marked block, nothing happens.
Move block	Edit | Cu**t**, Shift-Del Edit | **P**aste, Shift-Ins	Moves a previously marked block from its original position to the Clipboard and pastes it to the cursor position. The block disappears from its original position; the markers remain around the block at its new position. If no block is marked, nothing happens.
Delete block	Edit | **C**lear, *Ctrl-Del* *Ctrl-K Y*	Deletes a previously marked block. No provision exists to restore a deleted block, so be careful with this command.
Write block to disk	*Ctrl-K W*	Writes a previously marked block to a file. The block is left unchanged, and the markers remain in place. When you give this command, you are prompted for the name of the file to write to. The file can be given any legal name (the default extension is .C). If you prefer to use a file name without an extension, append a period to the end of its name. **Note:** You can use wildcards to select a file to overwrite; a directory is displayed. If the file specified already exists, a warning is issued before the existing file is overwritten. If no block is marked, nothing happens.
Read block from disk	*Ctrl-K R*	Reads a disk file into the current text at the cursor position, exactly as if it were a block. The text read is then marked as a block. When this command is issued, you are prompted for the name of the file to read. You can use wildcards to select a file to read; a directory is displayed. The file specified can be any legal file name.
Hide/display block	*Ctrl-K H*	Causes the visual marking of a block to be alternately switched off and on. The block manipulation commands (copy, move, delete, print, and write to a file) work only when the block is displayed. Block-related cursor movements (jump to beginning/end of block) work whether the block is hidden or displayed.
Print block	*Ctrl-K P*	Sends the marked block in the active Edit window to the printer.
Print	File | **P**rint	Sends the entire file in the active Edit window to the printer.

Other editing commands

The next table describes certain editing commands in more detail. The table is arranged alphabetically by the name of the command.

Table 5.3: Other editor commands in depth

Movement	**Command(s)**	**Function**
Autoindent	**O**ptions | **E**nvironment | **E**ditor	Opens the Editor Options dialog box, in which you can toggle the Autoindent Mode check box. Provides automatic indenting of successive lines. When Autoindent is active, the indentation of the current line is repeated on each following line; that is, when you press *Enter*, the cursor does not return to column one but to the starting column of the preceding non-empty line. When you want to change the indentation, use the *Spacebar* and ← key to select the new column. Autoindent is on by default.
Find place marker	*Ctrl-Q n*	Finds up to ten place markers (*n* can be any number in the range 0 to 9) in text. Move the cursor to any previously set marker by pressing *Ctrl-Q* and the marker number.
New file	**F**ile | **N**ew	Opens a new window.
Open file	**F**ile | **O**pen (*F3*)	Lets you load an existing file into an Edit window.
Quit edit	**F**ile | **Q**uit (*Alt-X*)	Quits Borland C++. You are asked whether you want to save the file to disk.
Undo changes	**E**dit | **U**ndo	Lets you undo editing changes.
Save file	**F**ile | **S**ave (*F2*)	Saves the file and returns to the editor.
Set place	*Ctrl-K n*	Mark up to ten places in text by pressing *Ctrl-K*, followed by a single marker digit (0 to 9). After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the *Ctrl-Q N* command (being sure to use the same marker number). You can have ten places marked in each window.
Tab	*Tab*	Tabs default to eight columns apart in the Borland C++ editor.
Tab mode	**O**ptions | **E**nvironment | **E**ditor	Opens the Editor Options dialog box, in which you can set the Use Tab Character check box. When the option is on, you can insert tab characters (ASCII character 9); when it's off, the tab is automatically inserted as the correct number of spaces.

Search and replace

The **Search | Find** and **Search | Replace** commands let you search for (and optionally replace) strings of up to 30 characters.

The search string is also called the target string.

The search string can contain any characters, including control characters. You can enter control characters with the *Ctrl-P* prefix. For example, enter a *Ctrl-T* by holding down the *Ctrl* key as you press *P* and then *T*. You can include a line break in a search string by specifying *Ctrl-M* (carriage return). (For searching regular expressions, take a look at the text file about GREP.)

The following sections list the steps for performing these operations.

Searching and searching again

1. Choose **Search | Find**. This opens the Find dialog box.
2. Type the string you are looking for (up to 30 letters) into the Text to Find input box.
3. You can also set various search options:
   - The Direction radio buttons control whether you do a forward or backward search.
   - The Scope radio buttons control how much of the file you search.
   - The Origin radio buttons control where the search begins.
   - The Options check boxes determine whether the search will be case sensitive for whole words only, and for regular expressions.

   Use *Tab* or your mouse to cycle through the options. Use ↑ and ↓ to set the radio buttons and *Space* to toggle the check boxes.
4. Finally, choose the OK button to carry out the search or the Cancel button to cancel. Borland C++ performs the operation.
5. If you want to search for the same item repeatedly, use **Search | Search Again**.

Search and replace

1. Choose **S**earch | **R**eplace. This opens the Replace dialog box.
2. Type the string you are looking for (up to 30 letters) into the Text to Find input box.
3. Press *Tab* or use your mouse to move to the New Text input box. Type in the replacement string.
4. You can then set the same search options as in the Find dialog box.
5. Finally, choose OK or Change All to begin the search, or choose Cancel to cancel. Borland C++ performs the operation. Choosing Change All will replace every occurrence found.
6. If you want to stop the operation, press *Esc* at any point when the search has paused.

Pair matching

There you are, debugging your source file that is full of functions, parenthesized expressions, nested comments, and a whole slew of other constructs that use delimiter pairs. In fact, your file is riddled with

- braces: { and }
- angle brackets: < and >
- parentheses: (and)
- brackets: [and]
- comment markers: /* and */
- double quotes: "
- single quotes: '

Finding the match to a particular paired construct can be tricky. Suppose you have a complicated expression with a number of nested expressions, and you want to make sure all the parentheses are properly balanced. Or say you're at the beginning of a function that stretches over several screens, and you want to jump to the end of that function. With Borland C++'s handy pair-matching commands, the solution is at your fingertips. Here's what you do:

1. Place the cursor on the delimiter in question (for example, the opening brace of some function that stretches for a couple of screens).
2. To locate the mate to this selected delimiter, simply press *Ctrl-Q [*. (In the example given, the mate should be at the end of the function.)
3. The editor immediately moves the cursor to the delimiter that matches the one you selected. If it moves to the one you had intended to be the mate, you know that the intervening code contains no unmatched delimiters of that type. If it moves to the wrong delimiter, you know there's trouble in River City; now all you need to do is track down the source of the problem.

We've told you the basics of Borland C++'s "Match Pair" commands; now you need some details about what you can and can't do with these commands, and notes about a few subtleties to keep in mind. This section covers the following points:

- There are actually two match pair editing commands: one for forward matching (*Ctrl-Q [*) and the other for backward matching (*Ctrl-Q]*).
- The way the editor searches for comment delimiters (/* and */) is slightly different from the way it performs the other searches.
- If there is no mate for the delimiter you've selected, the editor doesn't move the cursor.

Directional and nondirectional matching

Two match pair commands are necessary because some delimiters are *nondirectional.*

Opening braces and brackets and closing braces and parentheses are directional; the editor knows which way to search for the mate, so it doesn't matter which match pair command you give.

For example, suppose you tell the editor to find the match for an opening brace ({) or an opening bracket ([ ). The editor knows the matching delimiter can't be located *before* the one you've selected, so it searches forward for a match. If you tell the editor to find the mate to a closing brace (}) or a closing parenthesis ( )), it knows that the mate can't be located *after* the selected delimiter, so it automatically searches backward for a match.

However, if you tell the editor to find the match for a double quote (") or a single quote ('), it doesn't know automatically which way to go. You must specify the search

Double and single quotes are not directional. You must specify the correct match pair command.

direction by giving the correct match pair command. If you give the command *Ctrl-Q Ctrl-[*, the editor searches forward for the match; if you give the command *Ctrl-Q Ctrl-]*, it searches backward for the match.

The following table summarizes the delimiter pairs, whether they imply search direction, and whether they are nestable:

Table 5.4
Delimiter pairs

Nestable delimiters are explained after this table.

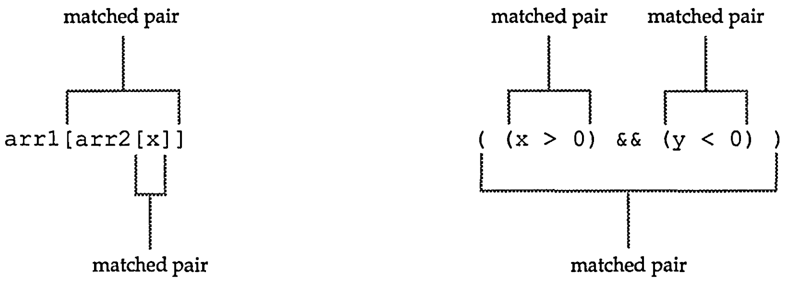
Delimiter pair	Direction implied?	Are they nestable?
{ }	Yes	Yes
( )	Yes	Yes
[ ]	Yes	Yes
< >	Yes	Yes
/* */	Yes	Yes and No
" "	No	No
' '	No	No

Nestable delimiters

Nestable means that, when the editor is searching for the mate to a directional delimiter, it keeps track of how many delimiter levels it enters and exits during the search.

This is best illustrated with some examples:

Figure 5.1
Search for match to square bracket or parenthesis



Comment delimiters

Because comment delimiters are two-character delimiters, you must take care when you highlight one for a match pair search. In either case, the editor recognizes only the *first* of the two characters: the slash (/) part of a /* comment delimiter, or the asterisk (*) part of a */ delimiter. If you place the cursor on the *second* character in either of these delimiters, the editor won't know what you're looking for, so it won't do any searching at all.

Also, as shown in Table 5.4, comment delimiters are sometimes nestable, sometimes not ("Yes and No"). This is not a vagary or an inability to decide: It is a test dependent on multiple conditions. ANSI-compatible C programs cannot contain nested comments, but Borland C++ provides an optional nested comments feature that you can set to on or off. This feature affects the nestability of comment delimiters when it comes to pair matching.

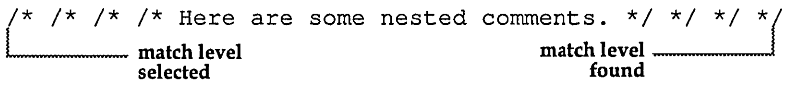
The search will be affected if unmatched delimiters of the same type in comments, quotes, or conditional compilation sections fall between the matched pair.

- If Nested Comments is checked, the editor treats comment delimiters as nestable and keeps track of the delimiter levels it enters and exits in the search for a match.
- If Nested Comments is unchecked, the editor won't treat comment delimiters as nestable; when a /* pair is selected, the first */ pair the editor finds is the match (and vice versa).

To set Nested Comments, choose **O**ptions | **C**ompiler | **S**ource. This opens the Source Options dialog box; use *Spacebar* to set the Nested Comments check box, then choose the OK button to confirm the setting.

Here are some examples to illustrate these differences. In the first two examples, the search is performed with *Ctrl-Q [*. In Figure 5.2, Nested Comments is checked. In Figure 5.3, Nested Comments is unchecked. In the third example, a backward search is performed using *Ctrl-Q]* with Nested Comments still unchecked.

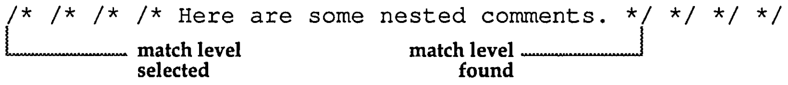
Figure 5.2
Forward search I



```
/* /* /* /* Here are some nested comments. */ */ */ */
```

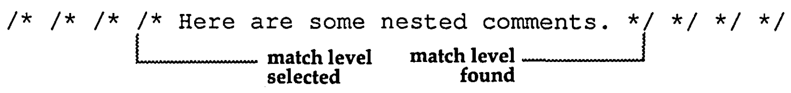
Note A backward search from the found */ will yield the selected /* when Nested Comments is checked.

Figure 5.3
Forward search II



```
/* /* /* /* Here are some nested comments. */ */ */ */
```

Figure 5.4
Backward search



```
/* /* /* /* Here are some nested comments. */ */ */ */
```

C H A P T E R

6

The command-line compiler

The command-line compiler lets you invoke all the functions of the IDE compiler from the DOS command line.

As an alternative to using the IDE, you can compile and run your programs with the command-line interface, hereafter referred to as the command-line compiler. The command-line compiler does more than just compile your files. It is in effect a command-line version of the IDE. Almost anything you can do within the IDE can also be done using the command-line compiler. You can set warnings on or off, use EMS or not, run in real or protected mode, invoke TASM (or another assembler) to assemble .ASM source files, and so on. In fact, to *compile only* you have to use the **–c** option at the command line.

This chapter is organized into two parts. The first describes how to use the command-line compiler and provides a summary table of all the options. The second part, starting on page 165, presents the options organized functionally (with groups of related options).

The summary table, Table 6.1 (starting on page 160), summarizes the command-line compiler options and provides a page-number cross-reference to where you can find more detailed information about each option. The options are also indexed individually so you can find discussions relating to them in other chapters and other books in the Borland C++ manuals.

Using the command-line compiler

You can run the command-line compiler in either real or protected mode. You can use protected mode if you have a 286, 386, or i486 machine with at least 640K conventional RAM and at least 576K extended or simulated expanded memory; most of the time you'll probably prefer to use it. Otherwise, use real mode.

Note that, although you may be running Borland C++ in protected mode, you are still generating applications to run in real mode. The greatest advantage to using Borland C++ in protected mode is that the compiler has *much* more room to run than if you were running it in real mode, while your application has more real-mode memory.

Running in real mode

To invoke Borland C++ from the command line in real mode, type BCC at the DOS prompt and follow it with a set of command-line arguments. Command-line arguments include compiler and linker options and file names. The generic command-line format is

You can also use a configuration file. See page 164 for details.

BCC [*option* [*option*...]] *filename* [*filename*...]

With two exceptions, each command-line option is preceded by a hyphen (–) and is separated from the BCC command, other options, and following file names by at least one space.

Running in protected mode

Running Borland C++ in protected mode involves interaction between three files: BCCX.EXE, BCCX.OVY, and TKERNEL.EXE. BCCX.EXE loads TKERNEL and BCCX.OVY, which is the protected-mode version of the command-line compiler. Although BCCX.EXE loads these files automatically, so that you don't need to be concerned with invoking them yourself, they do both need to be on the path or in the BCCX.EXE startup directory so it can find them.

Once you've verified that the paths are set correctly, running Borland C++ in protected mode is as simple as running it in real mode; the syntax is identical except for using BCCX in the place of BCC:

BCCX [*option* [*option...*]] *filename* [*filename...*]

The options and file names are identical to those for BCC; therefore, for the remainder of this chapter, when we mention the command-line compiler we mean both BCC and BCCX (unless we specifically state otherwise).

BCCX.EXE loads TKERNEL each time you invoke BCCX. You can save a great deal of loading time by preloading TKERNEL; before running BCCX, type

```
TKERNEL hi=yes
```

on the DOS command line. When you are through with your Borland C++ session, type

```
TKERNEL rem
```

to remove TKERNEL.

If you are using the command-line compiler in conjunction with Windows, you'll need to type

```
TKERNEL hi=yes kilos=1024
```

and invoke Windows using the **/s** (standard mode) option.

Using the options

The options are divided into three general types:

Compiler options are further divided into ten groups.

- compiler options, described starting on page 165
- linker options, described starting on page 183
- environment options, described starting on page 183

To see an onscreen list of the options, type `BCC` or `BCCX` (without any options or file names) at the DOS prompt. Then press *Enter.*

Use this feature to override settings in configuration files.

In order to select command-line options, enter a hyphen (–) immediately followed by the option letter (for example, **–I**). To turn an option off, add a second hyphen after the option letter. This is true for all toggle options (those that turn an option on or off): A trailing hyphen (–) turns the option off, and a trailing plus sign (+) or nothing turns it on. So, for example, **–C** and **–C+** both turn nested comments on, while **–C–** turns nested comments off.

Option precedence rules

The option precedence rules are simple; command-line options are evaluated from left to right, and the following rules apply:

- For any option that is *not* an **–I** or **–L** option, a duplication on the right overrides the same option on the left. (Thus an *off* option on the right cancels an *on* option to the left.)
- The **–I** and **–L** options on the left, however, take precedence over those on the right.

Table 6.1: Command-line options summary

Option	Page	Function
@filename	163	Gives the command-line compiler a response file name
+filename	164	Tell the command-line compiler to use the alternate configuration file *filename*
–1	167	Generate 80186 instructions
–1–	167	Generate 8088/8086 instructions
–2	167	Generate 80286 protected-mode compatible instructions
–A	174	Use only ANSI keywords
–A–, -AT	174	Use Borland C++ keywords (default)
–AK	174	Use only Kernighan and Ritchie keywords
–AU	174	Use only UNIX keywords
–a	167	Align word
–a–	167	Align byte (default)
–B	178	Compile and call the assembler to process inline assembly code
–b	168	Make enums word-sized (default)
–b–	168	Make enums signed or unsigned
–C	174	Nested comments on
–C–	174	Nested comments off (default)
–c	178	Compile to .OBJ but do not link
–D*name*	167	Define *name* to the string consisting of the null character
–D*name=string*	167	Defines *name* to *string*
–d	168	Merge duplicate strings on
–d–	168	Merge duplicate strings off (default)
–E*filename*	178	Use *filename* as the assembler to use
–e*filename*	183	Link to produce *filename*.EXE
–Fc	168	Generates COMDEFs
–Ff	168	Creates far variables automatically
–Ff=*size*	168	Creates far variables automatically; sets the threshold
–Fm	168	Enables the **–Fc**, **–Ff**, and **–Fs** options
–Fs	168	Assume DS = SS in all memory models
–f	169	Emulate floating point (default)
–f–	169	Don't do floating point
–ff	169	Fast floating point (default)
–ff–	169	Strict ANSI floating point
–f87	169	Use 8087 hardware instructions
–f287	169	Use 80287 hardware instructions

Table 6.1: Command-line options summary (continued)

Option	Page	Function
–G	172	Optimize for speed
–G–	172	Optimize for size (default)
–g*n*	174	Warnings: stop after *n* messages
–H	178	Causes the compiler to generate and use precompiled headers
–H–	178	Turns off generation and use of precompiled headers (default)
–Hu	178	Tells the compiler to use but not generate precompiled headers
–H=*filename*	178	Sets the name of the file for precompiled headers
–h	169	Use fast huge pointer arithmetic
–I*path*	184	Directories for include files
–i*n*	174	Make significant identifier length to be *n*
–j*n*	175	Errors: stop after *n* messages
–K	170	Default character type **unsigned**
–K–	170	Default character type **signed** (default)
–k	170	Standard stack frame on (default)
–L*path*	184	Directories for libraries
–l*x*	183	Pass option *x* to the linker (can use more than one *x*)
–l–*x*	183	Suppress option *x* for the linker
–M	183	Instruct the linker to create a map file
–mc	166	Compile using compact memory model
–mh	166	Compile using huge memory model
–ml	166	Compile using large memory model
–mm	166	Compile using medium memory model
–mm!	166	Compile using medium model; assume DS != SS
–ms	166	Compile using small memory model (default)
–ms!	166	Compile using small model; assume DS != SS
–mt	166	Compile using tiny memory model
–mt!	166	Compile using tiny model; assume DS != SS
–N	170	Check for stack overflow
–n*path*	184	Set the output directory
–O	172	Optimize jumps
–O–	172	No optimization (default)
–o*filename*	179	Compile source file to *filename*.obj
–P	179	Perform a C++ compile regardless of source file extension
–P*ext*	179	Perform a C++ compile and set the default extension to *ext*
–P–	179	Perform a C++ or C compile depending on source file extension (default)
–P–*ext*	179	Perform a C++ or C compile depending on extension; set default extension to *ext*
–p	170	Use Pascal calling convention
–p–	170	Use C calling convention (default)
–Qe	180	Instructs the compiler to use all available EMS memory (default)
–Qe–	180	Instructs the compiler to not use any EMS memory
–Qx	181	Instructs the compiler to use all available extended memory
–Qx=*nnnn*	181	Instructs the compiler to reserve *nnnn* Kbytes of extended memory for other programs, and to use the rest itself
–Qx=*nnnn,yyyy*	181	Instructs the compiler to reserve *nnnn* Kbytes of extended memory for other programs and *yyyy* for itself
–Qx=*,yyyy*	181	Instructs the compiler to reserve *yyyy* Kbytes of extended memory for itself
–Qx–	181	Instructs the compiler to not use any extended memory

Table 6.1: Command-line options summary (continued)

Option	Page	Function
–r	172	Use register variables on (default)
–r–	172	Suppresses the use of register variables.
–rd	173	Only allow declared register variables to be kept in registers
–S	179	Produce .ASM output file
–T*string*	179	Pass *string* as an option to TASM or assembler specified with **–E**
–T–	179	Remove all previous assembler options
–U*name*	167	Undefine any previous definitions of *name*
–u	170	Generate underscores (default)
–u–	170	Disables underscores
–V	182	Smart C++ virtual tables
–Vs	182	Local C++ virtual tables
–V0, –V1	182	External and Public C++ virtual tables
–Vf	182	Far C++ virtual tables
–v, –v–	171	Source debugging on
–vi, –vi–	172	Controls expansion of inline functions
–W	179	Creates an .OBJ for Windows with all functions exportable
–WD	180	Creates an .OBJ for Windows to be linked as a .DLL with all functions exportable
–WDE	180	Creates an .OBJ for Windows to be linked as a .DLL with explicit export functions
–WE	180	Creates an .OBJ for Windows with explicit export functions
–WS	180	Creates an .OBJ for Windows that uses smart callbacks
–w	175	Display warnings on
–w*xxx*	175	Enable *xxx* warning message
–w–*xxx*	175	Disable *xxx* warning message
–X	171	Disable compiler autodependency output
–Y	171	Enable overlay code generation
–Yo	171	Overlay the compiled files
–y	171	Line numbers on
–Z	173	Enable register usage optimization
–zA*name*	177	Code class
–zB*name*	177	BSS class
–zC*name*	177	Code segment
–zD*name*	177	BSS segment
–zE*name*	177	Far segment
–zF*name*	177	Far class
–zG*name*	177	BSS group
–zH*name*	177	Far group
–zP*name*	177	Code group
–zR*name*	178	Data segment
–zS*name*	178	Data group
–zT*name*	178	Data class
–zX*	178	Use default name for X. (default)

Syntax and file names

C++ files have the extension .CPP; see page 179 for information on changing the default extension.

Borland C++ compiles files according to the following set of rules:

filename.asm	Invoke TASM to assemble to .OBJ
filename.obj	Include as object at link time
filename.lib	Include as library at link time
filename	Compile FILENAME.CPP
filename.cpp	Compile FILENAME.CPP
filename.c	Compile FILENAME.C
filename.xyz	Compile FILENAME.XYZ

For example, given the following command line

```
BCC -a -f -C -O -Z -emyexe oldfile1 oldfile2 nextfile
```

Borland C++ compiles OLDFILE1.CPP, OLDFILE2.CPP, and NEXTFILE.CPP to an .OBJ, linking them to produce an executable program file named MYEXE.EXE with word alignment (**–a**), floating-point emulation (**–f**), nested comments (**–C**), optimization (**–O**), and register usage optimization (**–Z**) selected.

Borland C++ invokes TASM if you give it an .ASM file on the command line or if a .C or .CPP file contains inline assembly. The options that the command-line compiler gives to TASM are

/D__*MODEL*__ /D__*lang*__ /ml /*floatopt*

where *MODEL* is one of: TINY, SMALL, MEDIUM, COMPACT, LARGE, or HUGE. The **/ml** option tells TASM to assemble with case sensitivity on. *lang* is CDECL or PASCAL; *floatopt* is *r* when you've specified **–f87** or **–f287**; *e* otherwise.

Response files

Response files allow you to have longer command strings than DOS normally allows.

If you need to specify many options and/or files on the command line, you can place them in an ASCII text file, called a response file (you can of course name it anything you like). You can then tell the command-line compiler to read its command line from this file by including the appropriate file name prefixed with @. You can specify any number of such files, and you can mix them freely with other options and/or file names.

For example, suppose the file MOON.RSP contains STARS.C and RAIN.C. This command

```
BCC SUN.C @MOON.RSP ANYONE.C
```

will cause Borland C++ to compile the files SUN.C, STARS.C, RAIN.C, and ANYONE.C in real mode. It expands to

```
BCC SUN.C STARS.C RAIN.C ANYONE.C
```

Any options included in a response file are evaluated just as though they had been typed in on the command line. See page 160 for what those rules are.

Configuration files

TURBOC.CFG is not the same as TCCONFIG.TC, which is the default IDE version of a configuration file.

If you find you use a certain set of options over and over again, you can list them in a configuration file, called TURBOC.CFG by default. If you have a TURBOC.CFG configuration file, you don't need to worry about using it. When you run BCC (or BCCX), it automatically looks for TURBOC.CFG in the current directory. If it doesn't find it there *and* if you're running DOS 3.x or higher, Borland C++ then looks in the startup directory (where BCC.EXE or BCCX.EXE resides).

You can create more than one configuration file; each must have a unique name. To specify the alternate configuration file name, include its file name, prefixed with **+**, anywhere on the BCC (or BCCX) command line. For example, to read the option settings from the file D:\ALT.CFG, you could use the following command line:

```
BCC  +D:\ALT.CFG  ......
```

Your configuration file can be used in addition to or instead of options entered on the command line. If you don't want to use certain options that are listed in your configuration file, you can override them with options on the command line.

You can create the TURBOC.CFG file (or any alternate configuration file) using any standard ASCII editor or word processor, such as Borland C++'s integrated editor. You can list options (separated by spaces) on the same line or list them on separate lines.

Option precedence rules

In general, you can just remember that options given on the command line override the same options specified in the configuration file. This ability to override configuration file options with command-line options is an important one. If, for example, your configuration file contains several options, including the **–a** option (which you want to turn *off*), you can still use the configuration file but override the **–a** option by listing **–a–** in the command line. However, the rules are a little more detailed than that. The option precedence rules detailed on page 160 apply, with these additional rules:

1. When the options from the configuration file are combined with the command-line options, any **–I** and **–L** options in the configuration file are appended to the right of the command-line options. This means that the include and library directories specified in the command line are the first ones that Borland C++ searches (thereby giving the command-line **–I** and **–L** directories priority over those in the configuration file).
2. The remaining configuration file options are inserted immediately after the BCC (or BCCX) command (to the left of any command-line options). This gives the command-line options priority over the configuration file options.

Compiler options

Borland C++'s command-line compiler options fall into ten groups; the page references to the left of each group tell where you can find a discussion of each kind of option:

See page 166.

1. ***Memory model options*** let you tell Borland C++ which memory model to use when compiling your program.

See page 167.

2. ***Macro definitions*** let you define and undefine macros on the command line.

See page 167.

3. ***Code generation options*** govern characteristics of the generated code, such as the floating-point option, calling convention, character type, or CPU instructions.

See page 172.

4. ***Optimization options*** let you specify how the object code is to be optimized; for size or speed, with or without the use of register variables, and with or without assumptions about aliases.

See page 173.

5. ***Source code options*** cause the compiler to recognize (or ignore) certain features of the source code; implementation-specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths.

See page 174.

6. ***Error-reporting options*** let you tailor which warning messages the compiler will report, and the maximum number of warnings and errors that can occur before the compilation stops.

See page 177.

7. ***Segment-naming control options*** allow you to rename segments and to reassign their groups and classes.

See page 178.

8. ***Compilation control options*** let you direct the compiler to

   - compile to assembly code (rather than to an object module)
   - compile a source file that contains inline assembly (there are other ways though: use *#pragma inline* or just ignore it)
   - compile without linking
   - compile for Windows applications
   - use precompiled headers or not

See page 180.

9. ***EMS and extended memory options*** let you control how much expanded and extended memory Borland C++ uses.

See page 182.

10. ***C++ virtual table options*** let you control how virtual tables are handled.

Memory model

Memory model options let you tell Borland C++ which memory model to use when compiling your program. The memory models are: tiny, small, medium, compact, large, and huge.

See Chapter 6 in the Programmer's Guide for in-depth information on the memory models (what they are, how to use them).

–mc	Compile using compact memory model
–mh	Compile using huge memory model
–ml	Compile using large memory model
–mm	Compile using medium memory model
–mm!	Compile using medium model; DS != SS
–ms	Compile using small memory model (the default)
–ms!	Compile using small model; DS != SS
–mt	Compile using tiny memory model
–mt!	Compile using tiny model; DS != SS

You can't use the ***-N*** *option when using one of the DS != SS models.*

The net effect of the **–mt!**, **–ms!**, and **–mm!** options is actually very small. If you take the address of a stack variable (auto or parameter), the default (when DS == SS) is to make the resulting pointer a near (DS relative) pointer. In this way one can simply 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 a **_ss** pointer. But for the memory models affected, assigning the address to a near or default-sized pointer will produce a "Suspicious pointer conversion" warning. Such warnings are usually errors, and the warning defaults to on. You should regard this kind of warning as a likely error.

Macro definitions

Macro definitions let you define and undefine macros (also called *manifest* or *symbolic* constants) on the command line. The default definition is the null string. Macros defined on the command line override those in your source file.

–D*name*	Defines the named identifier *name* to the empty string.
–D*name=string*	Defines the named identifier *name* to the string *string* after the equal sign. *string* cannot contain any spaces or tabs.
–U*name*	Undefines any previous definitions of the named identifier *name*.

Borland C++ lets you make multiple **#define** entries on the command line in any of the following ways:

- You can include multiple entries after a single **–D** option, separating entries with a semicolon (this is known as "ganging" options):

  BCC –D*xxx;yyy*=1;zzz=NO MYFILE.C
- You can place more than one **–D** option on the command line:

  BCC –D*xxx* –D*yyy*=1 –Dzzz=NO MYFILE.C
- You can mix ganged and multiple **–D** listings:

  BCC –D*xxx* –D*yyy*=1;zzz=NO MYFILE.C

Code generation options

Code generation options govern characteristics of the generated code, such as the floating-point option, calling convention, character type, or CPU instructions.

–1	This option causes Borland C++ to generate extended 80186 instructions. It also generates 80286 programs running in real mode, such as with the IBM PC/AT under DOS.
–1–	Tells the compiler to generate 8088/8086 instructions.
–2	This option causes Borland C++ to generate 80286 protected-mode compatible instructions.

–a This option forces integer size and larger items to be aligned on a machine-word boundary. Extra bytes are inserted in a structure to ensure member alignment. 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. This option is off by default (**-a**–), allowing byte-wise alignment.

–b This option (which is on by default) tells the compiler to always allocate a whole word for enumeration types. (This is the way Turbo C 2.0 treats enumerations.)

–b– This option tells the compiler to allocate an unsigned or signed byte if the minimum and maximum values of the enumeration are both within the range of 0 to 255 or –128 to 127, respectively.

–d This option tells the compiler to merge literal strings when one string matches another, thereby producing smaller programs. This option is off by default (**-d**–).

–Fc This generates communal variables (COMDEFs) for global "C" variables that are not initialized and not declared as **static** or **extern**. The advantage of using this option is that header files that are included in several source files can contain declarations of global variables. So long as a given variable doesn't need to be initialized to a nonzero value, you don't need to include a definition for it in any of the source files. You can use this option when porting code that takes advantage of a similar feature with another implementation.

In the tiny, small, and medium models, the generated COMDEFs will be near; in the compact, large, and huge models they will be far.

–Ff When you use this option, global variables greater than or equal to the threshold size are automatically made far by the compiler. The threshold size defaults to 32,767; you can change it with the **–Ff=*size*** option. This option is useful for code that doesn't use the huge memory model but declares enough large global variables that their total size exceeds (or is close to) 64K.

–Ff= *size*	Use this option to change the threshold size used by the **–Ff** option.
–Fm	This option enables all the other **–F** options (**-Fc, –Ff** and **–Fs**). You can use it as a handy shortcut when porting code from other compilers.
–Fs	This option tells the compiler to assume that DS is equal to SS in all memory models; you can use it when porting code originally written for an implementation that makes the stack part of the data segment. When you specify this option, the compiler will link in an alternate startup module (C0Fx.OBJ) that will place the stack in the data segment.
–f	This option tells the compiler to emulate 80x87 calls at run time if the run-time system does not have an 80x87; if it does have one, the compiler calls the 80x87 chip for floating-point calculations (the default).
–f–	This option specifies that the program contains no floating-point calculations, so no floating-point libraries will be linked at the link step.
–ff	With this option, the compiler optimizes floating-point operations without regard to explicit or implicit type conversions. Answers can be faster than under ANSI operating mode. See Chapter 7, "Math," in the *Programmer's Guide* for details.
–ff–	This option turns off the fast floating-point option. The compiler follows strict ANSI rules regarding floating-point conversions.
–f87	This option tells the compiler to generate floating-point operations using inline 80x87 instructions rather than using calls to 80x87 emulation library routines. It specifies that a math coprocessor will be available at run time; therefore, programs compiled with this option will not run on a machine that does not have a math coprocessor.
–f287	This option is similar to **–f87,** but uses instructions that are only available with an 80287 (or higher) chip.
–h	This option offers an alternative way of calculating huge pointer expressions; a way which is much faster but must be used with caution. When you use this option, huge

pointers are normalized only when a segment wraparound occurs in the offset part. This will cause problems for huge arrays if any array elements cross a segment boundary. This option is off by default.

Normally, Borland C++ normalizes a huge pointer whenever adding to or subtracting from it. This ensures that, for example, if you have a huge array of **struct**s that's larger than 64K, indexing into the array and selecting a **struct** field will always work with **struct**s of any size. Borland C++ accomplishes this by always normalizing the results of huge pointer operations, so that the offset part contains a number that's no higher than 15. That way, 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.

–K This option tells the compiler to treat all **char** declarations as if they were **unsigned char** type. This allows for compatibility with other compilers that treat **char** declarations as **unsigned**. By default, **char** declarations are **signed** (**-K–**).

–k This option generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines. This option is on by default.

–N This option generates stack overflow logic at the entry of each function, which causes a stack overflow message to appear when a stack overflow is detected. This is costly in terms of both program size and speed but is provided as an option because stack overflows can be very difficult to detect. If an overflow is detected, the message "Stack overflow!" is printed and the program exits with an exit code of 1.

–p This option forces the compiler to generate all subroutine calls and all functions using the Pascal parameter-passing sequence. The resulting function calls are smaller and faster. Functions must pass the correct number and type of arguments, unlike normal C use, which permits a variable number of function arguments. You can use the **cdecl** statement to override this option and specifically declare functions to be C-type. This option is off by default (**-p–**).

–u With **–u** selected, when you declare an identifier, Borland C++ automatically puts an underscore (_) in front of the identifier before saving the identifier in the object module.

Borland C++ treats Pascal-type identifiers (those modified by the **pascal** keyword) differently—they are uppercase and are *not* prefixed with an underscore.

Unless you are an expert, don't use ***–u–****. See Chapter 9, "Interfacing with assembly language," in the Programmer's Guide for details about underscores.*

Underscores for C and C++ identifiers are optional, but on by default. You can turn them off with **–u–**. However, if you are using the standard Borland C++ libraries, you will encounter problems unless you rebuild the libraries. (To do this, you will need the Borland C++ run-time library source code; contact Borland for more information.)

–X This option disables generation of autodependency information in the output file. Modules compiled with this option enabled will not be able to use the autodependency feature of MAKE or of the IDE. Normally this option is only used for files that are to be put into .LIB files (to save disk space).

Note that you cannot use this option if you are using any of the ***–W*** *(Windows applications) options (and vice versa).*

–Y This option generates overlay-compatible code. Every file in an overlaid program must be compiled with this option; see Chapter 6, "Memory management," in the *Programmer's Guide* for details on overlays.

–Yo This option overlays the compiled file(s); see Chapter 6 in the *Programmer's Guide* for details.

–y This option includes line numbers in the object file for use by a symbolic debugger, such as Turbo Debugger. This increases the size of the object file but doesn't affect size or speed of the executable program. This option is useful only in concert with a symbolic debugger that can use the information. In general, **–v** is more useful than **–y** with Turbo Debugger.

The –v and –vi options

Turbo Debugger is both a source level (symbolic) and assembly level debugger.

–v This option tells the compiler to include debugging information in the .OBJ file so that the file(s) being compiled can be debugged with either Borland C++'s integrated debugger or the standalone Turbo Debugger. The compiler also passes this option on to the linker so it can include the debugging information in the .EXE file.

To facilitate debugging, this option also causes C++ inline functions to be treated as normal functions. If you want to avoid that, use **–vi**.

–vi	With this option enabled, C++ inline functions will be expanded inline.

In order to control the expansion of inline functions, the operation of the **–v** option is slightly different for C++. When inline function expansion is not enabled, the function will be generated and called like any other function. Debugging in the presence of inline expansion can be extremely difficult, so we provide the following options:

–v	This option turns debugging on and inline expansion off
–v–	This option turns debugging off and inline expansion on
–vi	This option turns inline expansion on
–vi–	This option turns inline expansion off

So, for example, if you want to turn both debugging and inline expansion on, you must use –v –vi.

Optimization options

Optimization options let you specify how the object code is to be optimized; for size or speed, with or without the use of register variables, and with or without assumptions about aliases.

–G	This option causes the compiler to bias its optimization in favor of speed over size.
–G–	This option, the default, causes the compiler to bias its optimization in favor of size over speed (where smaller is better).
–O	This option eliminates redundant jumps (such as jumps to jumps) and multiple copies of identical code that jump to the same location. It also suppresses redundant register loads. When **–Z** is not on, this will not change the behavior of your program (except, of course, that the code becomes more efficient).
–O–	When you disable optimizations, your code will compile very quickly but may be less efficient.
–r	This option enables the use of register variables (the default).

Unless you are an expert, don't use –r–.

–r– This option suppresses the use of register variables. When you are using this option, the compiler won't use register variables, and it won't preserve and respect register variables (SI,DI) from any caller. For that reason, you should not have code that uses register variables call code which has been compiled with **–r–**.

On the other hand, if you are interfacing with existing assembly-language code that does not preserve SI,DI, the **–r–** option allows you to call that code from Borland C++.

–rd This option only allows declared register variables to be kept in registers.

Exercise caution when using this option. The compiler cannot detect if a register has been invalidated indirectly by a pointer.

–Z This option allows the compiler to assume that variables are not accessed both directly and via a pointer in the same function. It only has an effect when used with **–O**.

The compiler keeps a table that reflects the current contents of registers. If a variable had to be loaded from memory into a register, the compiler remembers that the register now contains a copy of the variable. If the variable is used again, the compiler uses the copy in the register rather than the value in memory.

The **–Z** option determines how the compiler handles indirect assignments (that is, assignments via pointers, or assignments via reference in C++). Normally it assumes that such assignments could potentially change any variable. Therefore it has to forget about all copies of variables in registers (that is, erase the table). **–Z** tells the compiler that indirect assignments will not change variables, and that it is therefore safe to retain the copies.

The bottom line is that if you access a variable both directly and via a pointer within the same function, setting **–Z** *can* generate wrong code and is therefore unsafe to use. On the other hand, it *will* produce slightly faster code.

Source code options

Source code options cause the compiler to recognize (or ignore) certain features of the source code; implementation-specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths. These options are most significant if you plan to port your code to other systems.

–A This option compiles ANSI-compatible code: Any of the Borland C++ extension keywords are ignored and can be used as normal identifiers. These keywords include

See Chapter 1, "Lexical grammar," in the Programmer's Guide for a complete list of the Borland C++ keywords.

asm	**_ds**	**far**	**_loadds**
cdecl	**_es**	**huge**	**near**
_cs	**_export**	**interrupt**	**pascal**
			_ss

and the register pseudovariables, such as _AX, _BX, _SI, and so on.

–A– This option tells the compiler to use Borland C++ keywords. **–AT** is an alternate version of this option.

–AK This option tells the compiler to use only Kernighan and Ritchie keywords.

–AU This option tells the compiler to use only UNIX keywords.

–C This option allows you to nest comments. Comments may not normally be nested.

–i*n* This option causes the compiler to recognize only the first *n* characters of identifiers. All identifiers, whether variables, preprocessor macro names, or structure member names, are treated as distinct only if their first *n* characters are distinct.

By default, Borland C++ uses 32 characters per identifier. Other systems, including some UNIX compilers, ignore characters beyond the first eight. If you are porting to these other environments, you may wish to compile your code with a smaller number of significant characters. Compiling in this manner will help you see if there are any name conflicts in long identifiers when they are truncated to a shorter significant length.

Error-reporting options

Error-reporting options let you tailor which warning messages the compiler will report, and the maximum number of warnings and errors that can occur before the compilation stops.

–g*n*	This option tells Borland C++ to stop compiling after *n* warning messages.
–j*n*	This option tells the compiler to stop compiling after *n* error messages.
–w	This option causes the compiler to display warning messages. You can turn this off with **–w–**. You can enable or disable specific warning messages with **–w*xxx***, described in the following paragraphs.
–w*xxx*	This option enables the specific warning message indicated by *xxx*. The option **–w–*xxx*** suppresses the warning message indicated by *xxx*. The possible options for **–w*xxx*** are listed here and divided into four categories: ANSI violations, frequent errors (including more frequent errors), portability warnings, and C++ warnings. You can also use the pragma **warn** in your source code to control these options. See Chapter 4, "The preprocessor," in the *Programmer's Guide.*

For more information on these warnings, see Chapter 10, "Error messages," in the Programmer's Guide.

ANSI violations

The asterisk (*) indicates that the option is on by default. All others are off by default.

–wbbf	Bit fields must be **signed** or **unsigned int.**
–wbfs*	Untyped bit field assumed **signed int.**
–wbig*	Hexadecimal value contains more than three digits.
–wdcl*	Declaration does not specify a tag or an identifier.
–wdpu*	Declare *function* prior to use in prototype.
–wdup*	Redefinition of *macro* is not identical.
–weas	Assigning *integer_val* to *enumeration.*
–wext*	*Identifier* is declared as both external and static.
–wpin	This initialization is only partially bracketed.
–wret*	Both return and return with a value used.
–wstu*	Undefined structure *structure.*
–wsus*	Suspicious pointer conversion.
–wvoi*	Void functions may not return a value.
–wzdi*	Division by zero.

Frequent errors

–wamb	Ambiguous operators need parentheses.
–wamp	Superfluous & with function or array.
–wasc*	Restarting compile using assembly.
–wasm	Unknown assembler instruction.
–waus*	*Identifier* is assigned a value that is never used.

–wdef	Possible use of *identifier* before definition.
–weff*	Code has no effect.
–wfdt*	Function definition cannot be a typedef'd declaration.
–will*	Ill-formed pragma.
–wnod	No declaration for function *function*.
–wpar*	Parameter *parameter* is never used.
–wpia*	Possibly incorrect assignment.
–wpro	Call to function with no prototype.
–wrch*	Unreachable code.
–wrvl*	Function should return a value.
–wstv	Structure passed by value.
–wuse	*Identifier* declared but never used.

Portability warnings

–wcln	Constant is long.
–wcpt*	Nonportable pointer comparison.
–wrng*	Constant out of range in comparison.
–wrpt*	Nonportable pointer conversion.
–wsig	Conversion may lose significant digits.
–wucp	Mixing pointers to **signed** and **unsigned char.**

C++ warnings

–watt*	Assignment to **this** is obsolete; use **X::operator new** instead.
–wbei*	Initialization with inappropriate type.
–whid*	*Function1* hides virtual function *function2.*
–winl*	Functions containing *identifier* are not expanded inline.
–wlin*	Temporary used to initialize *identifier.*
–wlvc*	Temporary used for parameter in call to *identifier.*
–wncf*	Non-const function *function* called const object.
–wnci*	The constant member *identifier* is not initialized.
–wobi*	Base initialization without a class name is now obsolete.
–wofp*	This style of function definition is now obsolete.
–womf*	Obsolete syntax; use :: instead.
–wovl*	Use of overload is now unnecessary and obsolete.
–wscp*	*Identifier* is both a structure tag and a name; now obsolete.

Segment-naming control

Segment-naming control options allow you to rename segments and to reassign their groups and classes.

Don't use these options unless you have a good understanding of segmentation on the 8086 processor. Under normal circumstances, you will not need to specify segment names.

–zA*name*	This option changes the name of the code segment class to *name*. By default, the code segment is assigned to class CODE.
–zB*name*	This option changes the name of the uninitialized data segment class to *name*. By default, the uninitialized data segments are assigned to class BSS.
–zC*name*	This option changes the name of the code segment to *name*. By default, the code segment is named _TEXT, except for the medium, large and huge models, where the name is *filename_TEXT*. (*filename* here is the source file name.)
–zD*name*	This option changes the name of the uninitialized data segment to *name*. By default, the uninitialized data segment is named _BSS, except in the huge model, where no uninitialized data segment is generated.
–zE*name*	This option changes the name of the segment where far objects are put to *name*. By default, the segment name is the name of the far object followed by _FAR. A name beginning with an asterisk (*) indicates that the default string should be used.
–zF*name*	This option changes the name of the class for far objects to *name*. By default, the name is FAR_DATA. A name beginning with an asterisk (*) indicates that the default string should be used.
–zG*name*	This option changes the name of the uninitialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
–zH*name*	This option causes far objects to be put into group *name*. By default, far objects are not put into a group. A name beginning with an asterisk (*) indicates that the default string should be used.

See Chapter 6, "Memory management," in the Programmer's Guide for more on far objects.

–zP*name*	This option causes any output files to be generated with a code group for the code segment named *name*.
–zR*name*	This option sets the name of the initialized data segment to *name*. By default, the initialized data segment is named _DATA, except in the huge model, where the segment is named *filename*_DATA.
–zS*name*	This option changes the name of the initialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
–zT*name*	This option sets the name of the initialized data segment class to *name*. By default the initialized data segment class is named DATA.
–zX*	This option uses the default name for X. For example, **–zA*** assigns the default class name CODE to the code segment.

Compilation control options

Compilation control options allow you to control compilation of source files, such as whether your code is compiled as C or C++, whether to use precompiled headers, and what kind of Windows executable file is created. For more detailed information on how to create an Windows application, see Chapter 3.

See Appendix A for more on precompiled headers.

–B	This option compiles and calls the assembler to process inline assembly code.
–c	This option compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link command.
–E*filename*	This option uses *name* as the name of the assembler to use. By default, TASM is used.
–H	This option causes the compiler to generate and use precompiled headers.
–H–	This option turns off generation and use of precompiled headers (this is the default).
–Hu	This option tells the compiler to use but not generate precompiled headers.

–H=*filename*	This option sets the name of the file for precompiled headers. The default is TCDEF.SYM (located in the BCC or BCCX startup directory). This option also turns on generation and use of precompiled headers; that is, it also has the effect of **–H**.
–o*filename*	This option compiles the named file to the specified *filename*.obj.
–P	This option causes the compiler to compile your code as C++ always, regardless of extension. The compiler will assume that all files have .CPP extensions unless a different extension is specified with the code.
–P*ext*	This option causes the compiler to compile all files as C++; it changes the default extension to whatever you specify with *ext*. This option is available because some programmers use .C or another extension as their default extension for C++ code.
–P–	This option tells the compiler to compile a file as either C or C++, based on its extension. The default extension is .CPP. This option is the default.
–P–*ext*	This option also tells the compiler to compile code based on the extension (.CPP as C++ code, all other file-name extensions as C code). It further specifies what the default extension is to be.
–S	This option compiles the named source files and produces assembly language output files (.ASM), but does not assemble. When you use this option, Borland C++ will include the C or C++ source lines as comments in the produced .ASM file.
–T*string*	This option passes *string* as an option to TASM (or as an option to the assembler defined with **–E**).
–T–	This option removes all previously defined assembler options.

*Note that this option behaves differently from the **–P** option in Turbo C++ 1.x.*

*If you want to use your code written under Turbo C or Turbo C++ without having to think about file-name extensions, use either **–P–** or **–P–C**.*

*These five options (**-W**, **-WD**, **-WDE**, **-WE**, and **-WS**) relate to creating Windows applications. Note that you cannot use any of these options if you are using the **-Y** option (and vice versa).*

–W	This option creates the most general kind of Windows executable, although not necessarily the most efficient. The compiler generates export information for every far function. This does not mean that all far functions actually will be exported, it only means that export information is created for each far function. In order to actually export one of these functions, you must either use the **_export** keyword or add an entry for the function name in the EXPORTS section of the module definition file.
–WD	This option creates a module for use in a .DLL with all functions exportable.
–WDE	This option creates a module for use in a .DLL with only explicit functions exportable.
–WE	This option creates an object module with only function explicitly designated with **_export** as exportable.
–WS	This option creates an .OBJ with functions using smart callbacks. This option is recommended if you are writing Windows applications (*not* DLLs) which can assume SS = DS (most can). This option simplifies Windows programming; for instance, using it, you no longer need **MakeProcInstance** or **FreeProcInstance**, nor do you need to export your **WndProcs**; instead, you can directly call a **WndProc**. Enabling this option results in faster Windows executables.

Don't use this option for modules that will be compiled under the huge memory model.

EMS and extended memory options

If you have extended or expanded memory and you are not running the command-line compiler in protected mode, you may still want to have the compiler use all available memory. That's where these options come in.

–Qe	This option instructs the compiler to use all EMS memory it can find. This is on by default for the real-mode version of the command-line compiler (BCC). It speeds up your compilations, especially for large source files.

–Qe=*yyyy* This option instructs the compiler to use *yyyy* pages (in 16K page sizes) of EMS memory for itself.

–Qe– This option instructs the compiler not to use any EMS memory.

If you are in doubt about your systems' overall use of extended memory, don't use this option. Also, don't use this option when running BCCX.EXE.

–Qx This option instructs BCC to use all extended memory it can find. Like **–Qe**, this speeds up compilations of large source files. However, unlike **–Qe**, this option has to be used with care, because another program might be already using extended memory and not be recognized.

For example, using the VDISK RAM disk driver with this option is safe, while some disk caches are not.

–Qx=*nnnn* This option instructs the compiler to reserve *nnnn* Kbytes of extended memory for other programs and use the rest for itself. To figure out how much memory to reserve, you have to add up the memory that is used at the bottom of extended memory by resident programs like RAM disks or disk caches.

For example, if you use a disk cache, you might set it up so that it uses the first 512 Kbytes of extended memory. To tell the compiler to use the rest, you would specify `-Qx=512`.

If you aren't sure how much extended memory is used by resident utilities like RAM disks or disk caches, it is better not to use this option.

–Qx=*nnnn*,*yyyy* This option instructs the compiler to reserve *nnnn* Kbytes extended memory for other programs and *yyyy* Kbytes of extended memory for itself.

–Qx=,*yyyy* This option instructs the compiler to reserve *yyyy* Kbytes of extended memory for itself. The comma is essential.

–Qx– This option instructs the compiler not to use any extended memory. This is the default.

C++ virtual tables

The **–V** option controls the C++ virtual tables. There are five variations of the **–V** option:

–V	Smart C++ virtual tables
–Vs	Local C++ virtual tables
–V0	External C++ virtual tables
–V1	Public C++ virtual tables
–Vf	Far C++ virtual tables

–V Use this option when you want to generate C++ virtual tables (and inline functions not expanded inline) so that only one instance of a given virtual table or inline function will be included in the program. This produces the smallest and most efficient executables, but uses .OBJ and .ASM extensions only available with TLINK 3.0 and TASM 2.0 (or newer).

–Vs Use this option when you want Borland C++ to generate local virtual tables (and inline functions not expanded inline) such that each module gets its own private copy of each virtual table (or inline function) it uses. This option uses only standard .OBJ (and .ASM) constructs, but produces larger executables.

–V0, –V1 These options work together to create global virtual tables. If you don't want to use the Smart or Local options (**-V** or **–Vs**), you can use **–V0** and **–V1** to produce and reference global virtual tables. **–V0** generates external references to virtual tables; **–V1** produces public definitions for virtual tables.

When using these two options, at least one of the modules in the program must be compiled with the **–V1** option to supply the definitions for the virtual tables. All other modules should be compiled with the **–V0** option to refer to that Public copy of the virtual tables.

–Vf You can use this option independently of or in conjunction with any of the other virtual table options. It causes virtual tables to be created in the code segment instead of the data segment, and makes virtual table pointers into full 32-bit pointers (the latter is done automatically if you are using the huge memory model).

There are two primary reasons for using this option: to remove the virtual tables from the data segment, which may be getting full, and to be able to share objects (of classes with virtual functions) between modules that use different data segments (for example, a DLL and an executable using that DLL). You must compile all modules that may share objects either entirely with or entirely without this option. You can achieve the same effect by using the **huge** or **_export** modifiers on a class-by-class basis.

Linker options

See the section on TLINK in Chapter 7 for a list of linker options.

–efilename This option derives the executable program's name from *filename* by adding the file extension .EXE (the program name will then be *filename*.EXE). *filename* must immediately follow the **–e**, with no intervening whitespace. Without this option, the linker derives the .EXE file's name from the name of the first source or object file in the file name list. The default extension is .DLL when you are suing **–WD** or **–WDE**.

–l*x* This option (which is a lowercase l) passes option *x* to the linker. The option **–l–*x*** suppresses option *x*. More than one option can appear after the **–l**.

–M This option forces the linker to produce a full link map. The default is to produce no link map.

Environment options

When working with environment options, bear in mind that Borland C++ recognizes two types of library files: *implicit* and *user-specified* (also known as *explicit* library files). These are defined and discussed on page 185.

–I*path* This option (which is an uppercase I) causes the compiler to search *path* (the drive specifier or path name of a subdirectory) for include files (in addition to searching the standard places). A drive specifier is a single letter, either uppercase or lowercase, followed by a colon (:). A directory is any valid directory or directory path. You can use more than one **–I** directory option.

–L*path* This option forces the linker to get the C0x.OBJ start-up object file and the Borland C++ library files (Cx.LIB, CPx.LIB, MATHx.LIB, EMU.LIB, and FP87.LIB) from the named directory. By default, the linker looks for them in the current directory.

–n*path* This option places any .OBJ or .ASM files created by the compiler in the directory or drive named by *path*.

Searching for include and library files

Borland C++ can search multiple directories for include and library files. This means that the syntax for the library directories (**–L**) and include directories (**–I**) command-line options, like that of the **#define** option (**–D**), allows multiple listings of a given option.

Here is the syntax for these options:

Library directories:	–L*dirname*[;*dirname*;...]
Include directories:	–I*dirname*[;*dirname*;...]

The parameter *dirname* used with **–L** and **–I** can be any directory or directory path.

You can enter these multiple directories on the command line in the following ways:

- You can "gang" multiple entries with a single **–L** or **–I** option, separating ganged entries with a semicolon, like this:

```
BCC -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
BCC -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix ganged and multiple listings, like this:

```
BCC -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 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, or defines the specified constants, in order from left to right.

Note The IDE also supports multiple library directories through the "ganged entry" syntax.

File-search algorithms

The Borland C++ include-file search algorithms search for the **#include** files listed in your source code in the following way:

- If you put an `#include <somefile.h>` statement in your source code, Borland C++ searches for somefile.h only in the specified include directories.
- If, on the other hand, you put an `#include "somefile.h"` statement in your code, Borland C++ searches for somefile.h first in the current directory; if it does not find the header file there, it then searches in the include directories specified in the command line.

The library file search algorithms are similar to those for include files:

Your code written under any version of Turbo C or Turbo C++ will work without problems in Borland C++.

- ***Implicit libraries:*** Borland C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for **#include** <*somefile*.h>. [Implicit library files are the ones Borland C++ automatically links in. These are the C*x*.LIB and CWIN*x*.LIB files, EMU.LIB or FP87.LIB, MATH*x*.LIB, IMPORT.LIB, OVERLAY.LIB, and the start-up object files (C0*x*.OBJ, C0W*x*.OBJ, or C0D*x*.OBJ).]
- ***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 the 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++ searches for that library in the current directory first. Then (if the first search was unsuccessful), it looks in the specified library directories. This is similar to the search algorithm for **#include** "*somefile*.h".

- 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 the specified library directories.

An annotated example

Here is an example of a real-mode Borland C++ command line that incorporates multiple library and include directory options.

1. Your current drive is C:, and your current directory is C:\TURBOC, where BCC.EXE resides. Your A drive's current position is A:\ASTROLIB.
2. Your include files (.h or "header" files) are located in C:\TURBOC\INCLUDE.
3. Your startup files (C0T.OBJ, C0S.OBJ, ... , C0H.OBJ) are in C:\TURBOC.
4. Your standard Borland C++ library files (CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ... , EMU.LIB, FP87.LIB, and so forth) are in C:\TURBOC\LIB.
5. Your custom library files for star systems (which you created and manage with TLIB) are in C:\TURBOC\STARLIB. One of these libraries is PARX.LIB.
6. Your third-party-generated library files for quasars are in the A drive in \ASTROLIB. One of these libraries is WARP.LIB.

Under this configuration, you enter the following command:

```
BCC -mm -Llib;starlib -Iinclude orion.c umaj.c parx.lib a:\astrolib\warp.lib
```

Borland C++ compiles ORION.C and UMAJ.C to .OBJ files.

It then searches C:\TURBOC\INCLUDE for the include files in your source code, then links them with the medium model start-up code (C0M.OBJ), the medium model libraries (CM.LIB, MATHM.LIB), the standard floating-point emulation library (EMU.LIB), and the user-specified libraries (PARX.LIB and WARP.LIB), producing an executable file named ORION.EXE.

It searches for the startup code in C:\TURBOC (then stops because they're there); it searches for the standard libraries in C:\TURBOC\LIB (and stops because they're there).

When it searches for the user-specified library PARX.LIB, the compiler first looks in the current directory, C:\TURBOC. Not finding the library there, the compiler then searches the library directories in order: first C:\TURBOC\LIB, then C:\TURBOC\STARLIB (where it locates PARX.LIB).

Since an explicit path is given for the library WARP.LIB (A:\ASTROLIB\WARP.LIB), the compiler only looks there.

C H A P T E R

7

Utilities

Borland C++ comes with a host of powerful standalone utilities that you can use with your Borland C++ files or other modules. These are

- BGIOBJ, a conversion utility for graphics drivers and fonts (documented online)
- CPP, the preprocessor (documented online)
- GREP, a file-search utility (documented online)
- IMPDEF, which creates a module definition file
- IMPLIB, which generates an import library
- MAKE, the standalone program manager
- OBJXREF, an object module cross-referencer (documented online)
- PRJCFG, which updates options in a project file from a configuration file, or converts a project file to a configuration file (documented online)
- PRJCNVT, which converts Turbo C project files to the Borland C++ format (documented online)
- PRJ2MAK, which converts Borland C++ project files to MAKE files (documented online)
- THELP, the Turbo Help utility (documented online)
- TLIB, the Turbo Librarian
- TLINK, the Turbo Linker
- TOUCH, the file date and time changer (documented online)

- TRANCOPY, which copies transfer items from one project to another (documented online)
- TRIGRAPH, a character-conversion utility (documented online)

This chapter explains what IMPDEF, IMPLIB, MAKE, TLIB, and TLINK do, and illustrates, with code and command-line examples, how to use them. The rest of these utilities are documented in a text file called UTIL.DOC included with your distribution disks.

IMPDEF (module definition files)

An import library is used to provide access to a DLL's functions. See page 192 for more details.

IMPDEF works with IMPLIB to let you customize an import library to suit the needs of a specific application.

The syntax is

IMPDEF *DestName*.DEF *SourceName*.DLL

This creates a module definition file named *DestName*.DEF from the file *SourceName*.DLL. The module definition file would look something like this:

```
LIBRARY     FileName

DESCRIPTION 'Description'

EXPORTS
            ExportFuncName             @Ordinal
            ...
            ExportFuncName             @Ordinal
```

where *FileName* is the DLL's root filename, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement, *ExportFuncName* names an exported function and *Ordinal* is that function's ordinal value (an integer).

IMPDEF creates an editable source file that lists all of the export functions in the DLL. You can edit this .DEF file to contain only those functions that you would want to make available to a particular application, then run IMPLIB on the edited .DEF file. This process results in an import library that contains import information for a specific subset of a DLL's export functions.

For instance, let's say you're distributing a DLL that provides functions to be used by several applications. Every export function in the DLL is defined with **_export**. Now, if all of the applications used all of the DLL's exports, then you could simply use IMPLIB to make one import library for the DLL, and deliver that import library with the DLL. This import library would provide import information for all of the DLL's exports. The import library could be linked to any application, thus eliminating the need for the particular application to list every DLL function it uses in the IMPORTS section of its module definition file.

Now, let's say you want to give only a handful of the DLL's exports to a particular application. Ideally, you want a customized import library to be linked to that application—an import library that only provides import information for the subset of functions that the application will use. All of the other export functions in the DLL will be hidden to that client application.

To create an import library that satisfies these conditions, run IMPDEF on the compiled and linked DLL. IMPDEF produces a module definition file that contains an EXPORT section listing all of the DLL's export functions. You can edit that module definition file, removing EXPORTS section entries for those functions that you *don't* want in the customized import library. Once you've removed the exports that you don't want, run IMPLIB on the module definition file. The result will be an import library that contains import information for only those export functions listed in the EXPORTS section of the module definition file.

This utility is particularly handy for a DLL that uses C++ classes, for two reasons. First, if you use the **_export** keyword when defining a class, all of the non-inline member functions and static data members for that class are exported. It's easier to let IMPDEF make a module definition file for you because it lists all the exported functions, automatically including the member functions and static data members.

Since the names of these functions are mangled, it would be very tedious to list them all in the EXPORTS section of a module definition file simply so that you could create an import library from the module definition file. If you use IMPDEF to create the module definition file, it will include the ordinal value for each exported function, as well as that function's original name in a comment following the function entry, if the exported name is mangled. So, for instance, the module definition file created by IMPDEF for a DLL that used C++ classes would look something like this:

```
LIBRARY     FileName
DESCRIPTION 'Description'
EXPORTS
            MangledExportFuncName  @Ordinal ; ExportFuncName
            ...
            MangledExportFuncName  @Ordinal ; ExportFuncName
```

where *FileName* is the DLL's root filename, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement, *MangledExportFuncName* provides the mangled name, *Ordinal* is that function's ordinal value (an integer), and *ExportFuncName* gives the function's original name.

IMPLIB (import libraries)

The IMPLIB utility creates an import library that can be substituted for part or all of the IMPORTS section of a module definition file for a Windows application.

If a module uses functions from DLLs, you have two ways to tell the linker about them:

- You can add an IMPORTS section to the module definition file and list every function from DLLs that the module will use.
- Or you can include the import library for the DLLs when you link the module.

If you've created a Windows application, you've already used at least one import library, IMPORT.LIB. IMPORT.LIB is the import library for the Windows USER.EXE, GDI.EXE, and KERNEL.EXE DLLs. (IMPORT.LIB is linked automatically when you build a Windows application in the IDE; if you've used only the IDE to build applications, then you might not have seen it before.)

An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).

To create an import library for a DLL, type

```
IMPLIB Options LibName DefFiles Dlls
```

where *Options* is an optional list of one or more IMPLIB options, *LibName* (required) is the name for the new import library, *DefFiles* is a list of one or more existing module definition files for one or more DLLs, and *Dlls* is a list of one or more existing DLLs. You must specify at least one DLL or module definition file.

Table 7.1
IMPLIB options

You can use either a hyphen or a slash to precede IMPLIB's options.

Option	What it does
/i	Tells IMPLIB to ignore WEP, the Windows exit procedure required to end a DLL. Use this option if you are specifying more than one DLL on the IMPLIB command line.
Warning control:	
/t	Terse warnings.
/v	Verbose warnings.
/w	No warnings.

See page 190 for information on using IMPDEF and IMPLIB to customize an import library to suit the needs of a specific applicaiton.

Re-creating IMPORT.LIB

When Microsoft releases new versions of Windows you will probably need to replace the current version of IMPORT.LIB with a new one. The easiest way to do this is to build it yourself.

This command line builds the current version of IMPORT.LIB:

```
IMPLIB /I IMPORT.LIB GDI.EXE KERNEL.EXE USER.EXE KEYBOARD.DRV
          SOUND.DRV WIN87EM.DLL
```

If Windows is extended so that it uses additional DLLs, any new DLLs
will also have to appear on the command line.

MAKE: The program manager

Borland's command-line MAKE, derived from the UNIX program of the same name, helps you keep the executable versions of your programs current. Many programs consist of many source files, each of which may need to pass through preprocessors, assemblers, compilers, and other utilities before being combined with the rest of the program. Forgetting to recompile a module that has been changed—or that depends on something you've changed—

can lead to frustrating bugs. On the other hand, recompiling *everything* just to be safe can be a tremendous waste of time.

To find out how to create a makefile for Windows programs, see Chapter 3, page 109.

MAKE solves this problem. You provide MAKE with a description of how the source and object files of your program are processed to produce the finished product. MAKE looks at that description and at the date stamps on your files, then does what's necessary to create an up-to-date version. During this process, MAKE may invoke many different compilers, assemblers, linkers, and utilities, but it never does more than is necessary to update the finished program.

MAKE's usefulness extends beyond programming applications. You can use MAKE to control any process that involves selecting files by name and processing them to produce a finished product. Some common uses include text processing, automatic backups, sorting files by extension into other directories, and cleaning temporary files out of your directory.

How MAKE works

MAKE keeps your program up-to-date by performing the following tasks:

- Reads a special file (called a makefile) that you have created. This file tells MAKE which .OBJ and library files have to be linked in order to create your executable file, and which source and header files have to be compiled to create each .OBJ file.
- Checks the time and date of each .OBJ file against the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the source file must be recompiled.
- Calls the compiler to recompile the source file.
- Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of your executable file.
- If any of the .OBJ files is later than the .EXE file, calls the linker to recreate the .EXE file.

Caution!

MAKE relies completely upon the time stamp DOS places on each file. This means that, in order for MAKE to do its job, your system's time and date *must* be set correctly. If you own an AT or a PS/2, make sure that the battery is in good repair. Weak batteries can cause your system's clock to lose track of the date and time, and MAKE will no longer work as it should.

The original IBM PC and most compatibles didn't come with a built-in clock or calendar. If your system falls into this category, and you haven't added a clock, be sure to set the system time and date correctly (using the DOS DATE and TIME commands) each time you start your machine.

Starting MAKE

To use MAKE, type `make` at the DOS prompt. MAKE then looks for a file specifically named MAKEFILE. If MAKE can't find MAKEFILE, it looks for MAKEFILE.MAK; if it can't find that or BUILTINS.MAK (described later), it halts with an error message.

What if you want to use a file with a name other than MAKEFILE or MAKEFILE.MAK? You give MAKE the file (**-f**) option, like this:

```
MAKE -fMYFILE.MAK
```

The general syntax for MAKE is

make [*option* [*option*]] [*target* [*target* ...]]

where *option* is a MAKE option (discussed later), and *target* is the name of a target file to make.

Here are the MAKE syntax rules:

MAKE stops if any command it has executed is aborted via a Control-Break. Thus, a Control-Break stops the currently executing command and MAKE as well.

- The word *make* is followed by a space, then a list of make options.
- Each make option must be separated from its adjacent options by a space. Options can be placed in any order, and any number of these options can be entered (as long as there is room in the command line). All options that do not specify a string (**-s** or **–a**, for example) can have an optional – or + after them. This specifies whether you wish to turn the option off (–) or on (+).
- The list of MAKE options is followed by a space, then an optional list of targets.
- Each target must also be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, recompiling their constituents as necessary.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they will be built as necessary.

The BUILTINS.MAK file

You will often find that there are MAKE macros and rules that you use again and again. There are three ways of handling them.

- First, you can put them in every makefile you create.
- Second, you can put them all in one file and use the **!include** directive in each makefile you create. (See page 215 for more on directives.)
- Third, you can put them all in a BUILTINS.MAK file.

Each time you run MAKE, it looks for a BUILTINS.MAK file; however, there is no requirement that any BUILTINS.MAK file exist. If MAKE finds a BUILTINS.MAK file, it interprets that file first. If MAKE cannot find a BUILTINS.MAK file, it proceeds directly to interpreting MAKEFILE (or whatever makefile you specify).

The first place MAKE searches for BUILTINS.MAK is the current directory. If it's not there, *and* if you're running under DOS 3.0 or higher, MAKE then searches the directory from which MAKE.EXE was invoked. You should place the BUILTINS.MAK file in the same directory as the MAKE.EXE file.

MAKE always searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. Both BUILTINS.MAK and the makefile files have identical syntax rules.

MAKE also searches for any **!include** files (see page 217 for more on this MAKE directive) in the current directory. If you use the **–I** (include) option, it will also search in the directory specified with the **–I** option.

Command-line options

Here's a complete list of MAKE's command-line options. Note that case (upper or lower) *is* significant; the option **–d** is not a valid substitution for **–D**.

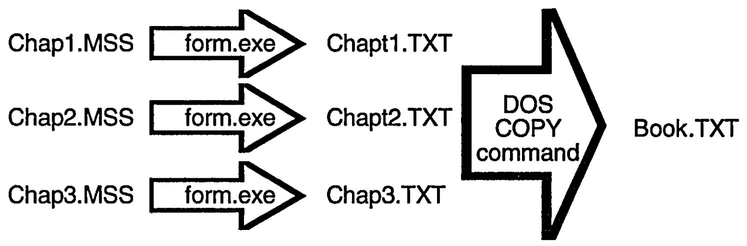
Table 7.2: MAKE options

Option	What it does
–? or **–h**	Prints a help message. The default options are displayed with plus signs following.
–a	Causes an automatic dependency check on .OBJ files.
–B	Builds all targets regardless of file dates.
–D*identifier*	Defines the named identifier to the string consisting of the single character 1 (one).
–D*iden=string*	Defines the named identifier *iden* to the string after the equal sign. The string cannot contain any spaces or tabs.
–f*filename*	Uses *filename* as the MAKE file. If *filename* does not exist and no extension is given, tries FILENAME.MAK.
–i	Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. This is equivalent to putting '–' in front of all commands in the MAKEFILE (described below).
–I*directory*	Searches for include files in the indicated directory (as well as in the current directory).
–K	Keeps (does not erase) temporary files created by MAKE. All temporary files have the form MAKE*nnnn*.$$$, where *nnnn* ranges from 0000 to 9999. See page 202 for more on temporary files.
–n	Prints the commands but does not actually perform them. This is useful for debugging a makefile.
–s	Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed.
–S	Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules.
–U*identifier*	Undefines any previous definitions of the named identifier.
–W	Writes the current specified non-string options (like **–s** and **–a**) to MAKE.EXE. (This makes them default.)

A simple use of MAKE

For our first example, let's look at a simple use of MAKE that doesn't involve programming. Suppose you're writing a book, and decide to keep each chapter of the manuscript in a separate file. (Let's assume, for the purposes of this example, that your book is quite short: It has three chapters, in the files CHAP1.MSS, CHAP2.MSS, and CHAP3.MSS.) To produce a current draft of the book, you run each chapter through a formatting program, called FORM.EXE, then use the DOS COPY command to concatenate the outputs to make a single file containing the draft, like this:

MAKE can also backup files, pull files out of different subdirectories, and even automatically run your programs should the data files they use be modified.



Like programming, writing a book requires a lot of concentration. As you write, you may modify one or more of the manuscript files, but you don't want to break your concentration by noting which ones you've changed. On the other hand, you don't want to forget to pass any of the files you've changed through the formatter before combining it with the others, or you won't have a fully updated draft of your book!

One inelegant and time-consuming way to solve this problem is to create a batch file that reformats every one of the manuscript files. It might contain the following commands:

```
FORM CHAP1.MSS
FORM CHAP2.MSS
FORM CHAP3.MSS
COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

Running this batch file would always produce an updated version of your book. However, suppose that, over time, your book got bigger and one day contained 15 chapters. The process of reformatting the entire book might become intolerably long.

MAKE can come to the rescue in this sort of situation. All you need to do is create a file, usually named MAKEFILE, which tells MAKE what files BOOK.TXT depends on and how to process them. This file will contain rules that explain how to rebuild BOOK.TXT when some of the files it depends on have been changed.

In this example, the first rule in your makefile might be

```
BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT
          COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

What does this mean? The first line (the one that begins with `book.txt:`) says that BOOK.TXT depends on the formatted text of each of the three chapters. If any of the files that BOOK.TXT depends on are newer than BOOK.TXT itself, MAKE must rebuild BOOK.TXT by executing the COPY command on the subsequent line.

This one rule doesn't tell the whole story, though. Each of the chapter files depends on a manuscript (.MSS) file. If any of the CHAP?.TXT files is newer than the corresponding .MSS file, the .MSS file must be recreated. Thus, you need to add more rules to the makefile as follows:

```
CHAP1.TXT: CHAP1.MSS
           FORM CHAP1.MSS

CHAP2.TXT: CHAP2.MSS
           FORM CHAP2.MSS

CHAP3.TXT: CHAP3.MSS
           FORM CHAP3.MSS
```

Each of these rules shows how to format one of the chapters, if necessary, from the original manuscript file.

MAKE understands that it must update the files that another file depends on before it attempts to update that file. Thus, if you change CHAP3.MSS, MAKE is smart enough to reformat Chapter 3 before combining the .TXT files to create BOOK.TXT.

We can add one more refinement to this simple example. The three rules look very much the same—in fact, they're identical except for the last character of each file name. And, it's pretty easy to forget to add a new rule each time you start a new chapter. To solve these problems, MAKE allows you to create something called an *implicit rule*, which shows how to make one type of file from another, based on the files' extensions. In this case, you can replace the three rules for the chapters with one implicit rule:

```
.MSS.TXT:
          FORM $*.MSS
```

This rule says, in effect, "If you need to make a .TXT file out of an .MSS file to make things current, here's how to do it." (You'll still have to update the first rule—the one that makes BOOK.TXT, so that MAKE knows to concatenate the new chapters into the output file. This rule, and others following, make use of a *macro*. See page 211 for an in-depth discussion of macros.)

Once you have the makefile in place, all you need to do to create an up-to-date draft of the book is type a single command at the DOS prompt: MAKE.

Creating makefiles

Creating a program from an assortment of program files, include files, header files, object files, and so on, is very similar to the text-processing example you just looked at. The main difference is that the commands you'll use at each step of the process will invoke preprocessors, compilers, assemblers, and linkers instead of a text formatter and the DOS COPY command. Let's explore how to create makefiles—the files that tell MAKE how to do these things—in greater depth.

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up-to-date. You can create as many makefiles as you want and name them whatever you want; MAKEFILE is just the default name that MAKE looks for if you don't specify a makefile when you run MAKE.

You create a makefile with any ASCII text editor, such as the IDE built-in editor, Sprint, or SideKick. All rules, definitions, and directives end at the end of a line. If a line is too long, you can continue it to the next line by placing a backslash (\) as the last character on the line.

Use whitespace (blanks and tabs) to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Components of a makefile

Creating a makefile is basically like writing a program, with definitions, commands, and directives. These are the constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules
- macro definitions
- directives:
  - file inclusion directives
  - conditional execution directives

- error detection directives
- macro undefinition directives

Let's look at each of these in more detail.

Comments

Comments begin with a pound sign (#) character; the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere; they don't have to start in a particular column.

A backslash will *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. If the backslash precedes the #, it is no longer the last character on the line; if it follows the #, then it is part of the comment itself.

Here are some examples of comments in a makefile:

Explicit and implicit rules are discussed following the section on commands.

```
# Makefile for my book

# This file updates the file BOOK.TXT each time I
# change one of the .MSS files

# Explicit rule to make BOOK.TXT from six chapters. Note the
# continuation lines.

BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT\
          CHAP4.TXT CHAP5.TXT CHAP6.TXT
          COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT+CHAP4.TXT+\
                  CHAP5.TXT+CHAP6.TXT BOOK.TXT

# Implicit rule to format individual chapters
.MSS.TXT:
          FORM $*.MSS
```

Command lists

Both explicit and implicit rules (discussed later) can have lists of commands. This section describes how these commands are processed by MAKE.

Commands in a command list take the form

> [*prefix* ...] *command_body*

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

Prefixes

The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at-sign (@) or a hyphen (–) followed immediately by a number.

Table 7.3
MAKE prefixes

Prefix	What it does
@	Prevents MAKE from displaying the command before executing it. The display is hidden even if the **–s** option is not given on the MAKE command line. This prefix applies only to the command on which it appears.
–num	Affects how MAKE treats exit codes. If a number (*num*) is provided, then MAKE aborts processing only if the exit status exceeds the number given. In this example, MAKE aborts only if the exit status exceeds 4: `-4 MYPROG SAMPLE.X` If no *–num* prefix is given and the status is nonzero, MAKE stops and deletes the current target file.
–	With a hyphen but no number, MAKE will not check the exit status at all. Regardless of the exit status, MAKE continues.

Exit codes are those returned by the executed commands (within the program).

Command body

The command body is treated exactly as if it were entered as a line to the DOS command line, with the exception that pipes (|) are not supported.

In addition to the **<**, **>**, and **>>** redirection operators, MAKE adds the **<<** and **&&** operators. These operators create a file on the fly for input to a command. The **<<** operator creates a temporary file and redirects the command's standard input so that it comes from the created file. If you have a program that accepted input from *stdin,* the command

```
 MYPROG <<!
This is a test
!
```

would create a temporary file containing the string "This is a test \n", redirecting it to be the sole input to *myprog*. The exclamation point (!) is a delimiter in this example; you can use any character except # or \ as a delimiter for the file. The first line containing the delimiter character as its first character ends the file. The rest of the line following the delimiter character (in this case, an exclamation point) is considered part of the preceding command.

The **&&** operator is similar to **<<**. It creates a temporary file, but instead of making the file the standard input to the command, the **&&** operator is replaced with the temporary file's name. This is useful when you want MAKE to create a file that's going to be used as input to a program. The following example creates a "response file" for TLINK:

Macros are covered starting on page 211.

```
MYPROG.EXE: $(MYOBJS)
    TLINK /c @&&!
COS $(MYOBJS)
$*
$*
$(MYLIBS) EMU.LIB MATHS.LIB CS.LIB
!
```

Note that macros (indicated by $ signs) are expanded when the file is created. The $* is replaced with the name of the file being built, without the extension, and $(MYOBJS) and $(MYLIBS) are replaced with the values of the macros MYOBJS and MYLIBS. Thus, TLINK might see a file that looks like this:

```
COS A.OBJ B.OBJ C.OBJ D.OBJ
MYPROG
MYPROG
W.LIB X.LIB Y.LIB Z.LIB EMU.LIB MATHS.LIB CS.LIB
```

All temporary files are deleted unless you use the **–K** command-line option. Use the **–K** option to "debug" your temporary files if they don't appear to be working correctly.

Batching programs

MAKE allows utilities that can operate on a list of files to be batched. Suppose, for example, that MAKE needs to submit several C files to Borland C++ for processing. MAKE could run BCC.EXE once for each file, but it's much more efficient to invoke BCC.EXE with a list of all the files to be compiled on the command line. This saves the overhead of reloading Borland C++ each time.

MAKE's batching feature lets you accumulate the names of files to be processed by a command, combine them into a list, and invoke that command only once for the whole list.

To cause MAKE to batch commands, you use braces in the command line:

command { *batch-item* } *...rest-of-command*

This command syntax delays the execution of the command until MAKE determines what command (if any) it has to invoke next. If the next command is identical except for what's in the braces, the two commands will be combined by appending the parts of the commands that appeared inside the braces.

Here's an example that shows how batching works. Suppose MAKE decides to invoke the following three commands in succession:

```
BCC {file1.c }
BCC {file2.c }
BCC {file3.c }
```

Rather than invoking Borland C++ three times, MAKE issues the single command

```
BCC file1.c file2.c file3.c
```

Note that the spaces at the ends of the file names in braces are essential to keep them apart, since the contents of the braces in each command are concatenated exactly as-is.

Here's an example that uses an implicit rule. Suppose your makefile had an implicit rule to compile C programs to .OBJ files:

```
.c.obj:
        BCC -c {$< }
```

As MAKE uses the implicit rule on each C file, it expands the macro **$<** into the actual name of the file and adds that name to the list of files to compile. (Again, note the space inside the braces to keep the names separate.) The list grows until one of three things happens:

- MAKE discovers that it has to run a program other than BCC
- there are no more commands to process
- MAKE runs out of room on the command line

If MAKE runs out of room on the command line, it puts as much as it can on one command line, then puts the rest on the next command line. When the list is done, MAKE invokes BCC (with the **–c** option) on the whole list of files at once.

Executing DOS commands

MAKE executes the DOS "internal" commands listed here by invoking a copy of COMMAND.COM to perform them:

break	del	path	set
cd	dir	prompt	time
chdir	echo	rd	type
cls	erase	rem	ver
copy	for	ren	verify
ctty	md	rename	vol
date	mkdir	rmdir	

MAKE searches for any other command name using the DOS search algorithm:

1. MAKE first searches for the file in the current directory, then searches each directory in the path.
2. In each directory, MAKE first searches for a file of the specified name with the extension .COM. If it doesn't find it, it searches for the same file name with an .EXE extension. Failing that, MAKE searches for a file by the specified name with a .BAT extension.
3. If MAKE finds a .BAT file, it invokes a copy of COMMAND.COM to execute the batch file.

If you supply a file-name extension in the command line, MAKE searches only for that extension. Here are some examples:

- This command causes COMMAND.COM to change the current directory to C:\include:

  ```
  cd c:\include
  ```

- MAKE uses the full search algorithm in searching for the appropriate files to perform this command:

  ```
  tlink lib\c0s x y,z,z,lib\cs
  ```

- MAKE searches for this file using only the .COM extension:

  ```
  myprog.com geo.xyz
  ```

- MAKE executes this command using the explicit file name provided:

  ```
  c:\myprogs\fil.exe -r
  ```

Explicit rules

The first rule in the example on page 201 is an explicit rule—a rule that specifies complete file names explicitly. Explicit rules take the form

```
target [target] ...: [source source ... ]
    [command]
    [command]
    ...
```

where *target* is the file to be updated, *source* is a file on which *target* depends, and *command* is any valid DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more source files, and an optional list of commands to be performed. Target and source file names listed in explicit rules can contain normal DOS drive and directory specifications; they can also contain wildcards.

Syntax here is important.

- *target* must be at the start of a line (in column 1).
- The *source* file(s) must be preceded by at least one space or tab, after the colon.
- Each *command* must be indented, (must be preceded by at least one blank or tab). As mentioned before, the backslash can be used as a continuation character if the list of source files or a given command is too long for one line.

Both the source files and the commands are optional; it is possible to have an explicit rule consisting only of *target [target ...]* followed by a colon.

The idea behind an explicit rule is that the command or commands listed will create or update *target,* usually using the *source* files. When MAKE encounters an explicit rule, it first checks to see if any of the *source* files are themselves target files elsewhere in the makefile. If so, MAKE evaluates that rule first.

Once all the *source* files have been created or updated based on other rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *source*. If any *source* has been modified more recently than *target,* the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or to the end of the file. Blank lines are ignored.

Special considerations

An explicit rule with no command lines following it is treated a little differently than an explicit rule with command lines.

- If an explicit rule includes commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on two sets of files: the files given in the explicit rule, and any file that matches an implicit rule for the target(s). This lets you specify a dependency to be handled by an implicit rule. For example,

  ```
  .c.obj
     BCC -c $<
  prog.obj:
  ```

  prog.obj depends on prog.c; MAKE executes the command line

  ```
  BCC -c prog.c
  ```

  if PROG.OBJ is out of date.

Examples

Here are some examples of explicit rules:

```
1. prog.exe: myprog.obj prog2.obj
     BCC myprog.obj prog2.obj
2. myprog.obj: myprog.c include\stdio.h
     BCC -c myprog.c
3. prog2.obj: prog2.c include\stdio.h
     BCC -c -K prog2.c
```

The three examples are from the same makefile. Only the modules affected by a change are rebuilt. If PROG2.C is changed, it's the only one recompiled; the same holds true for MYPROG.C. But if the include file stdio.h is changed, both are recompiled. (The link step is done if any of the .OBJ files in the dependency list have changed, which will happen when a recompile results from a change to a source file.)

Automatic dependency checking

Borland C++ works with MAKE to provide automatic dependency checking for include files. BCC and BC produce .OBJ files that tell MAKE what include files were used to create those .OBJ files. MAKE's **–a** command-line option checks this information and makes sure that everything is up-to-date.

When MAKE does an automatic dependency check, it reads the include files' names, times, and dates from the .OBJ file. The autodependency check will also work for include files inside of include files. If any include files have been modified, MAKE causes the .OBJ file to be recompiled. For example, consider the following explicit rule:

```
myprog.obj: myprog.c include\stdio.h
   BCC -c myprog.c
```

Now assume that the following source file, called MYPROG.C, has been compiled with BCC (version 2.0 or later):

```
#include <stdio.h>
#include "dcl.h"

void myprog() {}
```

If you then invoke MAKE with the following command line

```
make  -a  myprog.obj
```

it checks the time and date of MYPROG.C, and also of stdio.h and dcl.h.

Implicit rules

MAKE allows you to define *implicit* rules as well as explicit ones. Implicit rules are generalizations of explicit rules; they apply to all files that have certain identifying extensions.

Here's an example that illustrates the relationship between the two rules. Consider this explicit rule from the preceding example. The rule is typical because it follows a general principle: An .OBJ file is dependent on the .C file with the same file name and is created by executing BCC. In fact, you might have a makefile where you have several (or even several dozen) explicit rules following this same format.

By rewriting the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.c.obj:
   BCC -c $<
```

This rule means "Any file with the extension .C can be translated to a file of the same name with the extension .OBJ using this sequence of commands." The .OBJ file is created with the second line of the rule, where `$<` represents the file's name with the source (.C) extension. (The symbol `$<` is a special macro. Macros are discussed starting on page 211. The `$<` macro will be replaced by the full name of the appropriate .C source file each time the command executes.)

Here's the syntax for an implicit rule:

```
.source_extension.target_extension:
     [command]
     [command]
     ...
```

As before, the commands are optional and must be indented.

source_extension (which must begin with its period in column 1) is the extension of the source file; that is, it applies to any file having the format

fname.source_extension

Likewise, the *target_extension* refers to the file

fname.target_extension

where *fname* is the same for both files. In other words, this implicit rule replaces all explicit rules having the format

```
fname.target_extension: fname.source_extension
     [command]
     [command]
     ...
```

for any *fname*.

Note MAKE uses implicit rules if it can't find any explicit rules for a given target, or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is found with the same name as the target, but with the mentioned source extension.

For example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.c.obj:
   BCC -c $<
```

If you had a C program named RATIO.C that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
BCC -c ratio.c
```

which, of course, does the compile step necessary to create RATIO.OBJ.

MAKE also uses implicit rules if you give it an explicit rule with no commands. Suppose you had the following implicit rule at the start of your makefile:

```
.c.obj:
   BCC -c $<
```

You could then remove the command from the rule:

```
myprog.obj: myprog.c include\stdio.h
             BCC -c myprog.c
```

and it would execute exactly as before.

If you're using Borland C++ and you enable automatic dependency checking in MAKE, you can remove all explicit dependencies that have .OBJ files as targets. With automatic dependency checking enabled and implicit rules, the three-rule C example shown in the section on explicit rules becomes

```
.c.obj:
   BCC -c $<

prog.exe: myprog.obj prog2.obj
           tlink lib\c0s myprog prog2, prog, , lib\cs
```

You can write several implicit rules with the same target extension. If more than one implicit rule exists for a given target extension, the rules are checked in the order in which they appear in the makefile, until a match is found for the source extension, or until MAKE has checked all applicable rules.

MAKE uses the first implicit rule that involves a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule, up to the next line that begins without whitespace or to the end of the file, are considered to be part of the command list for the rule.

Macros

Often, you'll find yourself using certain commands, file names, or options again and again in your makefile. For instance, if you're writing a C program that uses the medium memory model, all your BCC commands will use the option **-mm**, which means to compile to the medium memory model. But suppose you wanted to switch to the large memory model. You could go through and change all the **-mm** options to **-ml**. Or, you could define a macro.

A *macro* is a name that represents some string of characters. A macro definition gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
MODEL = m
```

This line defines the macro `MODEL`, which is now equivalent to the string `m`. Using this macro, you could write each command to invoke the C compiler to look something like this:

```
BCC -c -m$(MODEL) myprog.c
```

When you run MAKE, each macro (in this case, `$(MODEL)`) is replaced with its expansion text (here, **m**). The command that's actually executed would be

```
BCC -c -mm myprog.c
```

Now, changing memory models is easy. If you change the first line to

```
MODEL = l
```

you've changed all the commands to use the large memory model. In fact, if you leave out the first line altogether, you can specify which memory model you want each time you run MAKE, using the **-D** (define) command-line option:

```
make -DMODEL = 1
```

This tells MAKE to treat **MODEL** as a macro with the expansion text *1*.

Defining macros

Macro definitions take the form

macro_name = expansion text

where *macro_name* is the name of the macro. *macro_name* should be a string of letters and digits with no whitespace in it, although you can have whitespace between *macro_name* and the equal sign (=). The *expansion text* is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by newline.

If *macro_name* has previously been defined, either by a macro definition in the makefile or by the **-D** option on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macro names **model**, **Model**, and **MODEL** are all different.

Using macros

You invoke macros in your makefile using this format

$(*macro_name*)

You need the parentheses for all invocations, even if the macro name is just one character long (with the exception of the predefined macros). This construct—`$(macro_name)`—is known as a *macro invocation*.

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Special considerations

Macros in macros: Macros cannot be invoked on the left side (*macro_name*) of a macro definition. They can be used on the right side (*expansion text*), but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

See page 215 for information on directives.

Macros in directives: Macro invocations are expanded immediately in **!if** and **!elif** directives. If the macro being invoked in an **!if** or **!elif** directive is not currently defined, it is expanded to the value 0 (FALSE).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

Predefined macros

MAKE comes with several special macros built in: **$d**, **$***, **$<**, **$:**, **$.**, and **$&**. The first is a test to see if a macro name is defined; it's used in the conditional directives **!if** and **!elif**. The others are file name macros, used in explicit and implicit rules. In addition, the current DOS environment strings (the strings you can view and set using the DOS SET command) are automatically loaded as macros. Finally, MAKE defines two macros: **__MSDOS__**, defined to be 1 (one); and **__MAKE__**, defined to be MAKE's version number in hexadecimal (for this version, 0x0300).

Table 7.4
MAKE macros

Macro	What it does
$d	Defined test macro
$*	Base file name macro with path
$<	Full file name macro with path
$:	Path only macro
$.	Full file name macro, no path
$&	Base file name macro, no path

Defined Test Macro ($d): The defined test macro (**$d**) expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in **!if** and **!elif** directives.

For example, suppose you want to modify your makefile so that if you don't specify a memory model, it'll use the medium one. You could put this at the start of your makefile:

```
!if !$d(MODEL)    # if MODEL is not defined
MODEL=m           # define it to m (MEDIUM)
!endif
```

If you then invoke MAKE with the command line

```
make -DMODEL=l
```

then **MODEL** is defined as *l*. If, however, you just invoke MAKE by itself,

```
make
```

then **MODEL** is defined as *m*, your "default" memory model.

File name macros

The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built.

Base file name macro ($*): The base file name macro is allowed in the commands for an explicit or an implicit rule. This macro (**$***) expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.C
$* expands to A:\P\TESTFILE
```

For example, you could modify this explicit rule

```
prog.exe: myprog.obj prog2.obj
          tlink lib\c0s myprog prog2, prog, , lib\cs
```

to look like this:

```
prog.exe: myprog.obj prog2.obj
          tlink lib\c0s myprog prog2, $*, , lib\cs
```

When the command in this rule is executed, the macro **$*** is replaced by the target file name without an extension and with a path. For implicit rules, this macro is very useful.

For example, an implicit rule might look like this:

```
.c.obj:
        BCC -c $*
```

Full file name macro ($<): The full file name macro (**$<**) is also used in the commands for an explicit or implicit rule. In an explicit rule, **$<** expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.C
$< expands to A:\P\TESTFILE.C
```

For example, the rule

```
mylib.obj: mylib.c
           copy $< \oldobjs
           BCC -c $*
```

copies MYLIB.OBJ to the directory \OLDOBJS before compiling MYLIB.C.

In an implicit rule, **$<** takes on the file name plus the source extension. For example, the implicit rule

```
.c.obj:
        BCC -c $*.c
```

produces exactly the same result as

```
.c.obj:
        BCC -c $<
```

because the extension of the target file name *must* be .C.

File-name path macro ($:): This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.C
$: expands to A:\P\
```

File-name and extension macro ($.): This macro expands to the file name, with an extension but without the path name, like this:

```
File name is A:\P\TESTFILE.C
$. expands to TESTFILE.C
```

File name only macro ($&): This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.C
$& expands to TESTFILE
```

Directives

Borland's MAKE allows something that other versions of MAKE don't: directives similar to those allowed in C, assembler, and Turbo Pascal. You can use these directives to perform a variety of useful and powerful actions. Some directives in a makefile begin with an exclamation point (!) as the first character of the line. Others begin with a period. Here is the complete list of MAKE directives:

Table 7.5
MAKE directives

.autodepend	Turns on autodependency checking.
!elif	Conditional execution.
!else	Conditional execution.
!endif	Conditional execution.
!error	Causes MAKE to stop and print an error message.
!if	Conditional execution.
.ignore	Tells MAKE to ignore return value of a command.

Table 7.5: MAKE directives (continued)

!include	Specifies a file to include in the makefile.
.noautodepend	Turns off autodependency checking.
.noignore	Turns off **.ignore**.
.nosilent	Tells MAKE to print commands before executing them.
.noswap	Tells MAKE to not swap itself in and out of memory.
.path.*ext*	Gives MAKE a path to search for files with extension *.EXT*.
.silent	Tells MAKE to not print commands before executing them.
.swap	Tells MAKE to swap itself in and out of memory.
!undef	Causes the definition for a specified macro to be forgotten.

Dot directives

Each of the following directives has a corresponding command-line option, but takes precedence over that option. For example, if you invoke MAKE like this:

```
make -a
```

but the makefile has a **.noautodepend** directive, then autodependency checking will be off.

.autodepend and **.noautodepend** turn on or off autodependency checking. They correspond to the **–a** command-line option.

.ignore and **.noignore** tell MAKE to ignore the return value of a command, much like placing the prefix – in front of it (described earlier). They correspond to the **–i** command-line option.

.silent and **.nosilent** tell MAKE whether or not to print commands before executing them. They correspond to the **–s** command-line option.

.swap and **.noswap** tell MAKE to swap itself out of memory. They correspond to the **–S** option.

.path.*ext*

This directive, placed in a makefile, tells MAKE where to look for files of the given extension. For example, if the following is in a makefile:

```
.path.c = C:\CSOURCE

.c.obj:
    BCC -c $*
```

```
tmp.exe: tmp.obj
  BCC tmp.obj
```

MAKE will look for TMP.C, the implied source file for TMP.OBJ, in C:\CSOURCE instead of the current directory.

The **.path** is also a macro that has the value of the path. The following is an example of the use of **.path**. The source files are contained in one directory, the .OBJ files in another, and all the .EXE files in the current directory.

```
.path.c   = C:\CSOURCE
.path.obj = C:\OBJS

.c.obj:
     BCC -c -o$(.path.obj)\$& $<

.obj.exe:
     BCC -e$&.exe $<

tmp.exe: tmp.obj
```

File-inclusion directive

A file-inclusion directive (**!include**) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

!include *"filename"*

You can nest these directives to any depth. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file MODEL.MAC that contained the following:

```
!if !$d(MODEL)
MODEL=m
!endif
```

You could use this conditional macro definition in any makefile by including the directive

```
!include "MODEL.MAC"
```

When MAKE encounters **!include**, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional execution directives

Conditional execution directives (**!if**, **!elif**, **!else**, and **!endif**) give you a measure of flexibility in constructing makefiles. Rules and macros can be made conditional, so that a command-line macro definition (using the **–D** option) can enable or disable sections of the makefile.

The format of these directives parallels those in C, assembly language, and Turbo Pascal:

```
!if expression
[ lines ]
!endif

!if expression
[ lines ]
!else
[ lines ]
!endif

!if expression
[ lines ]
!elif expression
[ lines ]
!endif
```

Note *[lines]* can be any of the following statement types:

- macro_definition
- explicit_rule
- implicit_rule
- include_directive
- if_group
- error_directive
- undef_directive

The conditional directives form a group, with at least an **!if** directive beginning the group and an **!endif** directive closing the group.

- One **!else** directive can appear in the group.
- **!elif** directives can appear between the **!if** and any **!else** directives.
- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested to any depth.

Any rules, commands, or directives must be complete within a single source file.

All **!if** directives must have matching **!endif** directives within the same source file. Thus the following include file is illegal, regardless of what's in any file that might include it, because it doesn't have a matching **!endif** directive:

```
!if $(FILE_COUNT) > 5
   some rules
!else
   other rules
<end-of-file>
```

Expressions allowed in conditional directives

Expressions are allowed in an **!if** or an **!elif** directive; they use a C-like syntax. The expression is evaluated as a simple 32-bit signed integer.

You can enter numbers as decimal, octal, or hexadecimal constants. If you know the C language, you already know how to write constants in MAKE; the formats are exactly the same. If you program in assembly language or Turbo Pascal, be sure to look closely at the examples that follow. These are legal constants in a MAKE expression:

```
4536   # decimal constant
0677   # octal constant (distinguished by leading 0)
0x23aF # hexadecimal constant (distinguished by leading 0x)
```

An expression can use any of the following operators:

Table 7.6
MAKE operators

See "Operator descriptions" in Chapter 1 of the Programmer's Guide for complete descriptions of these operators.

Operator	Operation	Operator	Operation
Unary operators		&	Bitwise AND
–	Negation	|	Bitwise OR
~	Bit complement	^	Bitwise XOR
!	Logical NOT	&&	Logical AND
		||	Logical OR
Binary operators		>	Greater than
+	Addition	<	Less than
–	Subtraction	>=	Greater than or equal
*	Multiplication	<=	Less than or equal
/	Division	==	Equality
%	Remainder	!=	Inequality
>>	Right shift	*Ternary operator*	
<<	Left shift	? :	Conditional expression

The operators have the same precedences as they do in the C language. Parentheses can be used to group operands in an expression. See the "Expressions" section in the *Programmer's Guide.*

You can invoke macros within an expression; the special macro **$d()** is recognized. After all macros have been expanded, the expression must have proper syntax.

Error directive

The error directive (**!error**) causes MAKE to stop and print a fatal diagnostic containing the text after **!error**. It takes the format

!error *any_text*

This directive is designed to be included in conditional directives to allow a user-defined error condition to abort MAKE. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(MODEL)
# if MODEL is not defined
!error MODEL not defined
!endif
```

If you reach this spot without having defined **MODEL**, then MAKE stops with this error message:

```
Fatal makefile 4: Error directive: MODEL not defined
```

Macro undefinition directive

The macro "undefinition" directive (**!undef**) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is

!undef *macro_name*

MAKE errors

MAKE diagnostic messages fall into two classes: errors and fatal errors.

- Errors indicate some sort of syntax or semantic error in the source makefile.
- When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart the compilation.

The following generic names and values appear in the messages listed in this section. When you get an error message, the appropriate name or value is substituted.

In manual	What you'll see onscreen
argument(s)	The command-line or other argument
expression	An expression
filename	A file name (with or without extension)
line number	A line number
message	A message string

Messages are listed in ASCII alphabetic order; messages beginning with symbols come first. Since messages that begin with one of the variables just listed cannot be alphabetized by what you will actually see when you receive such a message, all such messages have been placed at the beginning of each error message list.

For example, if you have tried to link a file named NOEXIT.C, you might receive the following actual message:

```
noexit does not exist--don't know how to make it
```

To look for this error, you would need to find

***filename* does not exist—don't know how to make it**

at the beginning of the list of error messages.

If the variable occurs later in the text of the error message (for example, "Illegal character in constant expression: *expression*"), you can find the explanation of the message in correct alphabetical order; in this case, under *I*.

Fatal error ***filename* does not exist – don't know how to make it**
There's a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.

Error **Bad file name format in include statement**
Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.

Error **Bad undef statement syntax**
An **!undef** statement must contain a single identifier and nothing else as the body of the statement.

Error **Character constant too long**
Character constants can be only one or two characters long.

Fatal error **Circular dependency exists in makefile**
The makefile indicates that a file needs to be up-to-date BEFORE it can be built. Take, for example, the explicit rules:

file*a*: file*b*
file*b*: file*c*
file*c*: file*a*

This implies that file*a* depends on file*b*, which depends on file*c*, and file*c* depends on file*a*. This is illegal, since a file cannot depend on itself, indirectly or directly.

Error **Command arguments too long**
The arguments to a command were more than the 127-character limit imposed by DOS.

Error **Command syntax error**
This message occurs if

- The first rule line of the makefile contained any leading whitespace.
- An implicit rule did not consist of *.ext.ext:*.
- An explicit rule did not contain a name before the : character.
- A macro definition did not contain a name before the = character.

Error **Command too long**
The length of a command has exceeded 128 characters. You might wish to use a response file.

Error **Division by zero**
A divide or remainder in an **!if** statement has a zero divisor.

Fatal error **Error directive: *message***
MAKE has processed an **#error** directive in the source file, and the text of the directive is displayed in the message.

Error **Expression syntax error in !if statement**
The expression in an **!if** statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

Error **File name too long**
The file name in an **!include** directive is too long for the compiler to process. File names in DOS can be no longer than 64 characters.

Error **If statement too long**
An **If** statement has exceeded 4,096 characters.

Error **Illegal character in constant expression <expression>**
MAKE encountered some character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.

Error **Illegal octal digit**
An octal constant was found containing a digit of 8 or 9.

Fatal error **Incorrect command-line argument: *argument***
You've used incorrect command-line arguments.

Error **Macro expansion too long**
A macro cannot expand to more than 4,096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

Error **Misplaced elif statement**
An **!elif** directive is missing a matching **!if** directive.

Error **Misplaced else statement**
There's an **!else** directive without any matching **!if** directive.

Error **Misplaced endif statement**
There's an **!endif** directive without any matching **!if** directive.

Error **No file name ending**
The file name in an include statement is missing the correct closing quote or angle bracket.

Fatal error **No terminator specified for in-line file operator**
The makefile contains either the **&&** or **<<** command-line operators to start an in-line file, but the file is not terminated.

Fatal error **Not enough memory**
All your working storage has been exhausted. You should perform your make on a machine with more memory. If you already have 640K in your machine, you may have to simplify the source file, or unload some memory-resident programs.

Error **Redefinition of target *filename***
The named file occurs on the left side of more than one explicit rule.

Error **Rule line too long**
An implicit or explicit rule was longer than 4,096 characters.

Fatal error **Unable to execute command**
A command failed to execute; this may be because the command file could not be found, or because it was misspelled, or (less likely) because the command itself exists but has been corrupted.

Error **Unable to open include file *filename***
The named file cannot be found. This can also be caused if an include file included itself. Check whether the named file exists.

Fatal error **Unable to open makefile**
The current directory does not contain a file named MAKEFILE, and there is no MAKEFILE.MAK.

Fatal error **Unable to redirect input or output**
Make was unable to open the temporary files necessary to redirect input or output. If you are on a network, make sure you have rights to the current directory.

Error **Unexpected end of file in conditional started on line *line number***
The source file ended before MAKE encountered an **!endif**. The **!endif** was either missing or misspelled.

Error **Unknown preprocessor statement**
A ! character was encountered at the beginning of a line, and the statement name following was not **error, undef, if, elif, include, else,** or **endif.**

TLIB: The Turbo Librarian

TLIB is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

The libraries included with Borland C++ were built with TLIB. You can use TLIB to build your own libraries, or to modify the Borland C++ libraries, your own libraries, libraries furnished by other programmers, or commercial libraries you have purchased. You can use TLIB to

- *create* a new library from a group of object modules
- *add* object modules or other libraries to an existing library
- *remove* object modules from an existing library
- *replace* object modules from an existing library
- *extract* object modules from an existing library
- *list* the contents of a new or existing library

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension.

TLIB can also create (and include in the library file) an Extended Dictionary, which may be used to speed up linking. See the section on the **/E** option (page 229) for details.

Although TLIB is not essential to creating executable programs with Borland C++, it is a useful programmer productivity tool. You will find TLIB indispensable for large development projects. If you work with object module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

Why use object module libraries?

When you program in C, you often create a collection of useful C functions, like the functions in the C run-time library. Because of C's modularity, you are likely to split those functions into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. If you always include all the source files, on the other hand, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of C functions. When you link your program with a library, the linker scans the library and automatically selects only those modules needed for the current program. In addition, a library consumes less disk space than a collection of object module files, especially if each of the object files is small. A library also speeds up the action of the linker, because it only opens a single file, instead of one file for each object module.

The TLIB command line

Run TLIB by typing a TLIB command line at the DOS prompt. To get a summary of TLIB's usage, just type `TLIB` and press *Enter.*

The TLIB command line takes the following general form, where items listed in square brackets ([*like this*]) are optional:

tlib *libname* [/C] [/E] [/P*size*] [*operations*] [, *listfile*]

This section summarizes each of these command-line components; the following sections provide details about using TLIB. For examples of how to use TLIB, refer to the "Examples" section on page 231.

Table 7.7: TLIB options

Option	Description
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a *libname*. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both BCC and BC's project-make facility require the .LIB extension in order to recognize library files. **Note:** If the named library does not exist and there are *add* operations, TLIB creates the library.
/C	The case-sensitive flag. This option is not normally used; see page 230 for a detailed explanation.
/E	Create Extended Dictionary; see page 229 for a detailed explanation.
/P*size*	Set the library page size to *size;* see page 229 for a detailed explanation.
operations	The list of operations TLIB performs. Operations may appear in any order. If you only want to examine the contents of the library, don't give any operations.
listfile	The name of the file listing library contents. The *listfile* name (if given) must be preceded by a comma. If you do not give a file name, no listing is produced. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the *listfile* is .LST. You can direct the listing to the screen by using the *listfile* name CON, or to the printer by using the name PRN.

The operation list

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character *action symbol* followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to the DOS-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

You can replace a module by first removing it, then adding the replacement module.

File and module names

TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you only need to supply the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB operations

TLIB recognizes three action symbols (–, +, *), which you can use singly or combined in pairs for a total of five distinct operations. For operations that use a pair of characters, the order of the characters is not important. The action symbols and what they do are listed here:

Table 7.8
TLIB action symbols

To create a library, add modules to a library that does not yet exist.

You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.

Action symbol	Name	Description
+	**Add**	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	**Remove**	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation only needs a module name. TLIB allows you to enter a full path name with drive and extension included, but ignores everything except the module name.
*	**Extract**	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.
-* *-	**Extract & Remove**	TLIB copies the named module to the corresponding file name and then removes it from the library. This is just a shorthand for an *extract* followed by a *remove* operation.
-+ +-	**Replace**	TLIB replaces the named module with the corresponding file. This is just a shorthand for a *remove* followed by an *add* operation.

Using response files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using *response files*. A response file is simply an ASCII text file (which can be created with the Borland C++ editor) that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one DOS command line.

To use a response file *pathname*, specify `@pathname` at any position on the TLIB command line. Note that the DOS path length limitation is 128 characters.

- More than one line of text can make up a response file; you use the "and" character (**&**) at the end of a line to indicate that another line follows.
- You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest.
- You can use more than one response file in a single TLIB command line.

See "Examples" for a sample response file and a TLIB command line incorporating it.

Creating an extended dictionary: The /E option

To speed up linking with large library files (such as the standard Cx.LIB library), you can direct TLIB to create an *extended dictionary* and append it to the library file. This dictionary contains, in a very compact form, information that is not included in the standard library dictionary. This information enables TLINK to process library files faster, especially when they are located on a floppy disk or a slow hard disk. All the libraries on your distribution disks contain the extended dictionary.

To create an extended dictionary for a library that is being modified, use the **/E** option when you invoke TLIB to add, remove, or replace modules in the library. To create an extended dictionary for an existing library that you don't want to modify, use the **/E** option and ask TLIB to remove a nonexistent module from the library. TLIB will display a warning that the specified module was not found in the library, but it will also create an extended dictionary for the specified library. For example, enter

```
tlib /E mylib -bogus
```

TLINK will ignore the debugging information in a library that has an extended dictionary, unless the **/e** option is used on the TLINK command line.

Setting the page size: The /P option

Every DOS library file contains a dictionary (which appears at the end of the .LIB file, following all of the object modules). For each module in the library, this dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library—it cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, the page size must be increased using the **/P** option; the page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), on the average 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use **/P** with the next available higher page size.

Advanced operation: The /C option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add to the library a module that would cause a duplicate symbol, TLIB displays a message and won't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates. Since C *does* treat uppercase and lowercase letters as distinct, use the **/C** option to add a module to a library that includes a symbol differing *only in case* from one already in the library. The **/C** option tells TLIB to accept a module with symbols in it that differ only in case from symbols already in the library.

*If you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should **not** use the /C option.*

It may seem odd that, without the **/C** option, TLIB rejects symbols that differ only in case, especially since C is a case-sensitive language. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case. Such linkers, for example, will treat *stars, Stars,* and *STARS* as the same identifier. TLINK, on the other hand, has no problem distinguishing uppercase and lowercase symbols, and it will properly accept a library containing symbols that differ only in case. In this example, then, Borland C++ would treat *stars, Stars,* and *STARS* as three separate identifiers. As long as you use the library only with TLINK, you can use the TLIB **/C** option without any problems.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB.

1. To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

   ```
   tlib mylib +x +y +z
   ```

2. To create a library as in #1 and get a listing in MYLIB.LST too, type

   ```
   tlib mylib +x +y +z, mylib.lst
   ```

3. To get a listing in CS.LST of an existing library CS.LIB, type

   ```
   tlib cs, cs.lst
   ```

4. To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

   ```
   tlib mylib -+x +a -z
   ```

5. To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type

   ```
   tlib mylib *y, mylib.lst
   ```

6. To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

   First create a text file, ALPHA.RSP, with

   ```
   +a.obj +b.obj +c.obj &
          +d.obj +e.obj +f.obj &
          +g.obj
   ```

   Then use the TLIB command, which produces a listing file named ALPHA.LST:

   ```
   tlib alpha @alpha.rsp, alpha.lst
   ```

TLINK (linker)

The IDE has its own built-in linker. When you invoke the command-line compiler BCC or BCCX, TLINK (for Turbo Linker) is invoked automatically unless you suppress the linking stage. If you suppress the linking stage, you must invoke TLINK manually. This section describes how to use TLINK or TLINKX (protected mode version) as a standalone linker. At the end of this section, you will find a complete list of linker messages generated by TLINK and by the built-in IDE linker.

By default, the command-line compiler calls TLINK when compilation is successful; TLINK then combines object modules and library files to produce the executable file.

Invoking TLINK

You can invoke TLINK at the DOS command line by typing `tlink` with or without parameters. When it is invoked without parameters, TLINK displays a summary of parameters and options. The following table briefly describes the TLINK options.

Note that this version of TLINK is sensitive to the case of its options; **/t** is not the same option as **/T**.

Table 7.9 TLINK options

You can use either a hyphen or a slash to precede TLINK's commands.

Option	What it does
/3	Enable 32-bit processing.
/A=*nnnn*	Specify segment alignment for NewExe (Windows) images.
/c	Treat case as significant in symbols.
/C	Treat EXPORTS and IMPORTS section of module definition file as case sensitive.
/d	Warn if duplicate symbols in libraries.
/e	Ignore Extended Dictionary.
/i	Initialize all segments.
/l	Include source line numbers.
/L	Specify library search paths.
/m	Create map file with publics.
/n	Don't use default libraries.
/o	Overlay following modules or libraries.
/P	Pack code segments.
/s	Create detailed map of segments.
/t	Generate .COM file. (Also /Tdc.)
/Td	Create target DOS executable.
/Tdc	Create target DOS .COM file.
/Tde	Create target DOS .EXE file.
/Tw	Create target Windows executable (.DLL or .EXE).
/Twe	Create target Windows application (.EXE).
/Twd	Create target Windows DLL (.DLL).
/v	Include full symbolic debug information.
/x	Don't create map file.
/ye	Use expanded memory for swapping.
/yx	Use extended memory for swapping.

The general syntax of a TLINK command line is

TLINK *objfiles, exefile, mapfile, libfiles, deffile*

This syntax specifies that you supply file names *in the given order*, separating the file *types* with commas.

An example of linking for DOS

If you supply the TLINK command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

TLINK will interpret it to mean that

- Case is significant during linking (**/c**).
- The .OBJ files to be linked are MAINLINE.OBJ, WD.OBJ, LN.OBJ, and TX.OBJ.
- The executable program name will be FIN.EXE.
- The map file is MFIN.MAP.
- The library files to be linked in are COMM.LIB and SUPPORT.LIB, both of which are in subdirectory WORK\LIB.
- No module definition file is specified.

An example of linking for Windows

To create a Windows application executable, you might use this command line:

```
tlink /Tw /c \BORLANDC\lib\c0ws winapp1 winapp2, winapp, winapp,
\BORLANDC\lib\cwins \BORLANDC\lib\cs \BORLANDC\lib\import, winapp.def
```

where

- The **/Tw** option tells TLINK to generate Windows executables.
- The **/c** option tells TLINK to be sensitive to case during linking. Note that the EXPORTS and IMPORTS sections in the module definition file will be still treated as case-insensitive unless the **/C** option is used.
- \BORLANDC\LIB\C0WS is the standard Windows initialization file and WINAPP1 and WINAPP2 are the module's object files; for all three files the .OBJ extension is assumed.
- WINAPP.EXE is the name of the target Windows executable.
- WINAPP.MAP is the name of the map file.
- \BORLANDC\LIB\CWINS is the small memory model runtime library for Windows, \BORLANDC\LIB\CS is the regular runtime library, and \BORLANDC\LIB\IMPORT is the library that provides access to the built-in Windows functions.
- WINAPP.DEF is the Windows module definition file used to specify additional link options.

File names on the TLINK command line

If you don't specify an executable file name, TLINK derives the name of the executable by appending .EXE or .DLL to the first object file name listed.

If you specify a complete file name for the executable file, TLINK will create the file with that name, but the actual nature of that executable depends on other options or on settings in the module definition file. For instance, if you specify WINAPP.EXE, but you provide the **/Twd** option, the executable will be created as a DLL but named WINAPP.EXE—probably not what you intended. Similarly, if you give WINAPP.DLL as the executable name, but include a **/Td** option on the command line, the file will be a DOS executable.

If no map file name is given, TLINK adds a .MAP extension to the .EXE file name. If no libraries are included, none will be linked. If you don't specify a module definition (.DEF) file *and* you have used the **/Tw** option, TLINK creates a Windows application based on default settings.

TLINK assumes or appends extensions to file names that have none:

- .OBJ for object files
- .EXE for executable files (when you use the **/t** or the **/Tdc** option, the executable file extension defaults to .COM rather than .EXE)
- .DLL for dynamic link libraries (when you use the /Twd option, or the /Tw option and the module definition file specifies a library)
- .MAP for map files
- .LIB for library files
- .DEF for module definition files.

All of the file names *except* object files are optional. So, for instance,

```
TLINK dosapp dosapp2
```

links the files DOSAPP.OBJ and DOSAPP2.OBJ, creates a DOS executable file called DOSAPP.EXE, creates a map file called DOSAPP.MAP, links no libraries, and uses no module definition file.

Using response files

Because TLINK is now sensitive to the case of its options, you may need to modify response files, batch files, and makefiles intended for a previous version of TLINK.

TLINK lets you supply the various parameters on the command line, in a response file, or in any combination of the two.

A response file is just a text file that contains the options and file names that you would usually type in after the name TLINK on your command line.

Unlike the command line, however, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line. When a plus occurs at the end of a line but it immediately follows one of the TLINK options that uses + to enable the option (such as **/ye+**), the + is not treated as a line continuation character.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

with the following response file, FINRESP:

```
/c mainline wd+
   ln tx,fin
   mfin
   work\lib\comm work\lib\support
```

You would then enter your TLINK command as

```
tlink @finresp
```

Note that you must precede the file name with an "at" character (@) to indicate that the next name is a response file.

Alternately, you may break your link command into multiple response files. For example, you can break the previous command line into the following two response files:

File name	Contents
LISTOBJS	`mainline+` `wd+` `ln tx`
LISTLIBS	`lib\comm+` `lib\support`

You would then enter the TLINK command as

```
tlink /c @listobjs,fin,mfin,@listlibs
```

The TLINK configuration file

The command line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded (DOS 3.0 or higher).

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked.

For instance, the following TLINK.CFG file

```
/Lc:\BORLANDC\lib;c:\winapps\lib
/v /s
/Tw
```

tells TLINK to search the specified directories for libraries, include debug information, create a detailed segment map, and produce a Windows program.

Using TLINK with Borland C++ modules

Borland C++ supports six different memory models: tiny, small, compact, medium, large, and huge. When you create an executable Borland C++ file using TLINK, you must include the initialization module and libraries for the memory model being used.

The general format for linking Borland C++ programs with TLINK is

tlink C0[W | D]*x* *myobjs*, *exe*,[*map*], [IMPORT] [*mylibs*] [OVERLAY] [CWIN*x*] [EMU | FP87 math*x*] C*x*, [*deffile*]

where

myobjs	=	the .OBJ files you want linked, specify path if not in current directory
exe	=	the name to be given the executable file
[*map*]	=	the name to be given the map file (optional)

[*mylibs*]	=	the library files you want included at link time (optional), specify path if not in current directory, or use **/L** option to specify search paths
deffile	=	the module definition file for a Windows executable

Be sure to include paths for the startup code and libraries (or use the **/L** option to specify a list of search paths for startup and library files). The other file names on this general TLINK command line represent Borland C++ files, as follows:

*If you are using the tiny model and you want TLINK to produce a .COM file, you must also specify the **/t** or **/Tdc** option.*

C0*x* | C0F*x* | C0W*x* | C0D*x* |	=	initialization module for DOS executable, DOS executable written for another compiler, Windows application, or Windows DLL (choose one) with memory model t (DOS only), s, c, m, l, or h (DOS only)
IMPORT	=	Windows import library; the library that provides access to the built-in Windows functions
OVERLAY	=	overlay manager library; needed only for overlaid programs (not compatible with Windows)
CWIN*x*	=	run-time library for executable under Windows with memory model s, c, m, or l
EMU | FP87	=	the floating-point libraries (choose one)
MATH*x*	=	math library for memory model s, c, m, l, or h
C*x*	=	run-time library for memory model s, c, m, l, or h

Startup code

The initialization modules have the name C0*x*.OBJ, C0W*x*.OBJ, or C0D*x*.OBJ (for DOS, a Windows application, and a Windows DLL, respectively), where *x* is a single letter corresponding to the model: *t* for tiny (DOS only), *s* for small, *c* for compact, *m* for medium, *l* for large, and *h* for huge (DOS only).

New! The C0F*x*.OBJ modules are provided for compatibility with source files intended for compilers from other vendors. The C0F*x*.OBJ modules substitute for the C0*x*.OBJ modules; they are to be linked with DOS applications only, not Windows applications or DLLs. These initialization modules alter the memory model so that the stack segment is inside the data segment. The appropriate

C0F*x*.OBJ module will be used automatically if you use either the **–Fs** or the **–Fm** command-line compiler option.

Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, that no stack has been created, or that fixup overflows occurred.

The initialization module must also appear as the first object file in the list. The initialization module arranges the order of the various segments of the program. If it is not first, the program segments may not be placed in memory properly, causing some frustrating program bugs.

Be sure that you give an explicit name for the executable file name on the TLINK command line. Otherwise, your program name will be something like C0*x*.EXE—probably not what you wanted!

Libraries

The order of objects and libraries is very important. You must always put the Borland C++ start-up module (C0*x*.OBJ, C0F*x*, C0W*x*.OBJ, or C0D*x*.OBJ) first in the list of objects. Then, the library list should contain, in this specific order:

- your own libraries (if any)
- if you want to overlay your program (DOS only), you must include OVERLAY.LIB; this library must precede the C*x*.LIB library
- CWIN*x*.LIB if you want your program to run under Windows
- if you are using floating point math, FP87.LIB or EMU.LIB (required for DOS only), followed by MATH*x*.LIB (required for DOS and Windows)
- C*x*.LIB (Borland C++ run-time library file for DOS or Windows)

If you want to create a Windows application or DLL you must link IMPORT.LIB to provide access to the built-in Windows functions. IMPORT.LIB can be included anywhere in the list. If you are using any Borland C++ graphics functions, you must link in GRAPHICS.LIB anywhere in the list. The graphics library is independent of memory models, but is for DOS only (not Windows).

If your program uses any floating-point, you must include a math library (MATH*x*.LIB) in the link command. For DOS applications (but not for Windows applications or DLLs), you will also need to include either the EMU.LIB or FP87. LIB floating-point libraries. Borland C++'s two floating-point libraries are independent of the program's memory model.

- Use EMU.LIB if you want to include floating-point emulation logic. With EMU.LIB the program will work on machines whether they have a math coprocessor (80x87) chip or not.
- If you know that the program will always be run on a machine with a math coprocessor chip, the FP87.LIB library will produce a smaller and faster executable program.

The math libraries have the name MATH*x*.LIB, where *x* is a single letter corresponding to the model: s, c, m, l, h (the tiny and small models share the library MATHS.LIB).

You can always include the emulator (DOS only) and math libraries in a link command line. If you do so, and if your program does no floating-point work, nothing from those libraries will be added to your executable program file. However, if you know there is no floating-point work in your program, you can save some time in your links by excluding those libraries from the command line.

You must always include the C run-time library for the program's memory model. The C run-time libraries have the name C*x*.LIB, where *x* is a single letter corresponding to the model, as before. Use the same C run-time library for both DOS and Windows executables.

If you aren't going to use all six memory models, and your hard disk space is limited, you may want to keep only the files for the model(s) you are using. Here's a list of the library files needed for each memory model (you'll also need FP87.LIB or EMU.LIB for DOS only, and IMPORT.LIB for Windows):

Table 7.10 DOS application .OBJ and .LIB files

Model	Regular Startup Module	Compatibility Startup Module	Math Library	Run-time Library
Tiny	C0T.OBJ	C0FT.OBJ	MATHS.LIB	CS.LIB
Small	C0S.OBJ	C0FS.OBJ	MATHS.LIB	CS.LIB
Compact	C0C.OBJ	C0FC.OBJ	MATHC.LIB	CC.LIB
Medium	C0M.OBJ	C0FM.OBJ	MATHM.LIB	CM.LIB
Large	C0L.OBJ	C0FL.OBJ	MATHL.LIB	CL.LIB
Huge	C0H.OBJ	C0FH.OBJ	MATHH.LIB	CH.LIB

Table 7.11
Windows application .OBJ and .LIB files

Model	Startup for applications	Windows RTL	Math Library	RTL
Small	C0WS.OBJ	CWINS.LIB	MATHS.LIB	CS.LIB
Compact	C0WC.OBJ	CWINC.LIB	MATHC.LIB	CC.LIB
Medium	C0WM.OBJ	CWINM.LIB	MATHM.LIB	CM.LIB
Large	C0WL.OBJ	CWINL.LIB	MATHL.LIB	CL.LIB

Note that the tiny and small models use the same libraries, but have different startup files (C0T.OBJ vs. C0S.OBJ).

Table 7.12
DLL object and library files

Model	Startup for DLLs	Windows RTL	Math Library	RTL
Small	C0DS.OBJ	CWINC.LIB	MATHS.LIB	CS.LIB
Compact	C0DC.OBJ	CWINC.LIB	MATHC.LIB	CC.LIB
Medium	C0DM.OBJ	CWINL.LIB	MATHM.LIB	CM.LIB
Large	C0DL.OBJ	CWINL.LIB	MATHL.LIB	CL.LIB

See Chapter 3, page 123 for more information on DLLs.

Using TLINK with BCC

You can also use BCC, the standalone Borland C++ compiler, as a "front end" to TLINK that will invoke TLINK with the correct startup file, libraries, and executable program name.

See Chapter 6, "The command-line compiler," for more on BCC.

To do this, you give file names on the BCC command line with explicit .OBJ and .LIB extensions. For example, given the following BCC command line,

```
BCC -MX MAINFILE.OBJ SUB1.OBJ MYLIB.LIB
```

BCC will invoke TLINK with the files C0*x*.OBJ, EMU.LIB, MATH*x*.LIB and C*x*.LIB (initialization module, default 8087 emulation library, math library and run-time library for memory model *x*). TLINK will link these along with your own modules MAINLINE.OBJ and SUB1.OBJ, and your own library MYLIB.LIB.

To compile and link a Windows program, include one of the **–W** options on the compiler command-line, as well as any other options. The compiler will take care of linking in C0W*x*.OBJ, CWIN*x*.LIB, and IMPORT.LIB.

When BCC invokes TLINK, it uses the **/c** (case-sensitive link) option by default. You can override this default with **–l –c**).

TLINK options

TLINK options can occur anywhere on the command line. The options consist of a slash (/), a hyphen (–), or the DOS switch character, followed by the option. (The DOS switch character is / by default. You can change it by using an INT 21H call.)

Important! Starting with this version of TLINK, the case of an option is significant (/t is not the same as /T). All options that existed before this version are lowercase. If you used uppercase options in response files or make files intended for a previous version of TLINK, you'll need to modify those files before you use them with this version.

If you have more than one option, spaces are not significant (**/m/c** is the same as **/m /c**), and you can have them appear in different places on the command line. The following sections describe each of the options.

The TLINK configuration file

The command-line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded (DOS 3.0 or higher).

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked. Whitespace is ignored.

For instance, the following TLINK.CFG file

```
/Lc:\BORLANDC\lib;c:\winapps\lib
/v /s
/Tw
```

tells TLINK to search the specified directories for libraries, include debug information, create a detailed segment map, and produce a Windows program.

/3 (80386 32-bit code)

The **/3** option should be used when one or more of the object modules linked has been produced by TASM or a compatible assembler, and contains 32-bit code for the 80386 processor. This option increases the memory requirements of TLINK and slows down linking, so it should be used only when necessary.

/A (align segments)

The **/A** option specifies a byte value on which to align segments. Segments smaller than the specified value will be padded up to the value. The syntax is

/A=*nnnn*

where *nnnn* is a number which respresents the alignment factor. *nnnn* must be a power of two. For instance, `/A=16` indicates that segments should be aligned on a paragraph boundary.

The default segment alignment size is 512. For efficiency, you should use the smallest value that still allows for correct segment offsets in the segment table. The file addresses in the segment table are multiplied by the alignment factor in order to be used as byte offsets into the executable file. Since the offsets are stored as 16-bit words, 65536 times the alignment factor is the limit of segment offsets that can be represented in the segment table. If you get this message, increase the segment alignment value.

/c (case sensitivity)

The **/c** option forces the case to be significant in public and external symbols.

/C (case sensitive exports)

By default, TLINK treats the EXPORTS and IMPORTS sections of the module definition file as case-insensitive. The **/C** or **/C+** option turns on case-sensitivity; **/C-** turns off case-sensitivity.

/d (duplicate symbols)

Normally, TLINK will not warn you if a symbol appears in more than one library file. If the symbol must be included in the program, TLINK will use the copy of that symbol in the first file on the command line in which it is found. Since this is a commonly used feature, TLINK does not normally warn about the duplicate symbols. The following hypothetical situation illustrates how you might want to use this feature.

Suppose you have two libraries: one called SUPPORT.LIB, and a supplemental one called DEBUGSUP.LIB. Suppose also that DEBUGSUP.LIB contains duplicates of some of the routines in SUPPORT.LIB (but the duplicate routines in DEBUGSUP.LIB include slightly different functionality, such as debugging versions of the routines). If you include DEBUGSUP.LIB *first* in the link command, you will get the debugging routines and *not* the routines in SUPPORT.LIB.

If you are not using this feature or are not sure which routines are duplicated, you may include the **/d** option. TLINK will list all symbols duplicated in libraries, even if those symbols are not going to be used in the program.

Given this option, TLINK will also warn about symbols that appear both in an .OBJ and a .LIB file. In this case, since the symbol that appears in the first (left-most) file listed on the command line is the one linked in, the symbol in the .OBJ file is the one that will be used.

With Borland C++, the distributed libraries you would use in any given link command do not contain any duplicated symbols. So while EMU.LIB and FP87.LIB (or CS.LIB and CL.LIB) obviously have duplicate symbols, they would never rightfully be used together in a single link. There are no symbols duplicated between EMU.LIB, MATHS.LIB, and CS.LIB, for example.

/e (no extended dictionary)

The library files that are shipped with Borland C++ all contain an *extended dictionary* with information that enables TLINK to link faster with those libraries. This extended dictionary can also be added to any other library file using the **/E** option with TLIB (see the section on TLIB starting on page 225). The TLINK **/e** option disables the use of this dictionary.

Although linking with libraries that contain an extended dictionary is faster, you might want to use the **/e** option if you have a program that needs slightly more memory to link when an extended dictionary is used.

Unless you use **/e** to turn off extended dictionary use, TLINK will ignore any debugging information contained in a library that has an extended dictionary.

/i (uninitialized trailing segments)

The **/i** option causes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This option is not normally necessary.

/l (line numbers)

The **/l** option creates a section in the .MAP file for source code line numbers. To use it, you must have created the .OBJ files by compiling with the **-y** (Line numbers...On) or -v (Debug information) option. If you use the **/x** to tell TLINK to create no map at all, this option will have no effect.

/L (library search paths)

The **/L** option lets you specify a list of directories that TLINK searches for libraries if an explicit path is not specified. TLINK searches the current directory before those specified with the **/L** option. For example,

```
TLINK /Lc:\BORLANDC\lib;c:\mylibs splash logo,,,utils .\logolib
```

With this command line, TLINK first searches the current directory for UTILS.LIB, then searches C:\BORLANDC\LIB and C:\MYLIBS. Because .\LOGOLIB explicitly names the current directory, TLINK does not search the libraries specified with the **/L** option to find LOGOLIB.LIB.

TLINK also searches for the C or C++ initialization module (C0*x*.OBJ, C0W*x*.OBJ, C0D*x*.OBJ) on the specified library search path.

/m, /s, and /x (map options)

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error messages produced during the link. If you don't want to create a map, turn it off with the **/x** option.

If you want to create a more complete map, the **/m** option will add a list of public symbols to the map file, sorted alphabetically as well as in increasing address order. This kind of map file is useful in debugging. Many debuggers can use the list of public symbols to allow you to refer to symbolic addresses when you are debugging.

The **/s** option creates a map file with segments, public symbols and the program start address just like the **/m** option did, but also adds a detailed segment map. The following is an example of a detailed segment map:

Figure 7.1
Detailed map of segments

```
Address    Length  Class   Segment Name  Group     Module   Alignment/
           (Bytes)                                          Combining

0000:0000  0E5B   C=CODE  S=SYMB_TEXT   G=(none)  M=SYMB.C  ACBP=28
00E5:000B  2735   C=CODE  S=QUAL_TEXT   G=(none)  M=QUAL.C  ACBP=28
0359:0000  002B   C=CODE  S=SCOPY_TEXT  G=(none)  M=SCOPY   ACBP=28
035B:000B  003A   C=CODE  S=LRSH_TEXT   G=(none)  M=LRSH    ACBP=20
035F:0005  0083   C=CODE  S=PADA_TEXT   G=(none)  M=PADA    ACBP=20
0367:0008  005B   C=CODE  S=PADD_TEXT   G=(none)  M=PADD    ACBP=20
036D:0003  0025   C=CODE  S=PSBP_TEXT   G=(none)  M=PSBP    ACBP=20
036F:0008  05CE   C=CODE  S=BRK_TEXT    G=(none)  M=BRK     ACBP=28
03CC:0006  066F   C=CODE  S=FLOAT_TEXT  G=(none)  M=FLOAT   ACBP=20
0433:0006  000B   C=DATA  S=_DATA       G=DGROUP  M=SYMB.C  ACBP=48
0433:0012  00D3   C=DATA  S=_DATA       G=DGROUP  M=QUAL.C  ACBP=48
0433:00E6  000E   C=DATA  S=_DATA       G=DGROUP  M=BRK     ACBP=48
0442:0004  0004   C=BSS   S=_BSS        G=DGROUP  M=SYMB.C  ACBP=48
0442:0008  0002   C=BSS   S=_BSS        G=DGROUP  M=QUAL.C  ACBP=48
0442:000A  000E   C=BSS   S=_BSS        G=DGROUP  M=BRK     ACBP=48
```

For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Except for the ACBP field, the information in the detailed segment map is self-explanatory.

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, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (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.
The C field (combination)	00	May not be combined.
	08	A public combining segment.
The B field (big)	00	Segment less than 64K.
	02	Segment exactly 64K.

When you request a detailed map with the **/s** option, the list of public symbols (if it appears) has public symbols flagged with "idle" if there are no references to that symbol. 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:

```
0C7F:031E   idle   Symbol1
0000:3EA2          Symbol2
0C7F:0320   idle   Symbol3
```

/n (ignore default libraries)

The **/n** option causes the linker to ignore default libraries specified by some compilers. You may want to use this option when linking modules written in another language.

/o (overlays)

The **/o** option causes the code in all modules or libraries specified after the option to be overlaid. It remains in effect until the next comma (explicit or implicit) or **/o–** in the command stream. **/o–** turns off overlaying. (Chapter 6, "Memory management," in the *Programmer's Guide* covers overlays in more detail.)

The **/o** option can be optionally followed by a segment class name; this will cause all segments of that class to be overlaid. When no such name is specified, all segments of classes ending with CODE will be overlaid. Multiple **/o** options can be given, thus overlaying segments of several classes; all **/o** options remain in effect until the next comma or **/o–** is encountered.

The syntax **/o**#*xx*, where *xx* is a two-digit hexadecimal number, overrides the overlay interrupt number, which by default is 3FH.

Here are some examples of **/o** options:

Table 7.13
TLINK overlay options

Option	Result
/o	Overlay all code segments until next comma or **/o–**.
/o–	Stop overlaying.
/oOVY	Overlay segments of class OVY until the next comma or **/o–**.
/oCODE /oOVLY	Overlay segments of class CODE or class OVLY until next comma or **/o–**.
/o#F0	Use interrupt vector 0F0H for overlays.

If you use the **/o** option, it will be turned off automatically before the libraries are processed. If you want to overlay a library, you must use another **/o** right before all the libraries or right before the library you want to overlay.

You can't use the **/o** option with any **/Tw** option; Windows applications can't be overlaid. However, in order to achieve essentially the same results under Windows, use discardable code segments (see page 250 for information on defining code segments attributes in the module definition file).

/P (pack code segments)

When you use **/P**, when TLINK links Windows executables, TLINK combines as many code segments as possible in one physical segment up to the code segment packing limit. Code segment packing never creates segments greater than this limit; TLINK starts a new segment if it needs to.

The default code segment packing limit is 8,192 bytes (8K). To change it, use

```
/P=n
```

where *n* specifies the number of bytes between 1 and 65,536. You would probably want the limit to be a multiple of 4K under 386 enhanced mode.

Although the optimum segment size in 386 enhanced mode is 4K, the default code segment packing size is 8K. Because typical code segments are likely to be from 4K to 8K, an 8K packing size will probably result in more effective packing.

Because there is a certain amount of system overhead for every segment maintained, code segment packing, by reducing the number of segments to maintain, typically increases performance. The **/P** option turns code packing on; it is off by default. **/P-** turns off code segment packing (useful if you've turned it on in the configuration file and want to disable it for a particular link).

/t (tiny model .COM file)

If you compile your file in the tiny memory model and link it with this option toggled on, TLINK will generate a .COM file instead of the usual .EXE file. Also, when you use **/t**, the default extension for the executable file is .COM. This works the same as the **/Tdc** option. Neither **/t** or **/Tdc** is compatible with the Windows option, **/Tw**.

Note: .COM files may not exceed 64K in size, cannot have any segment-relative fixups, cannot define a stack segment, and must have a starting address equal to 0:100H. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100H.

TLINK can't generate debugging information for a .COM file. If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **–c** option; this creates a .COM file from and .EXE.

/Td and /Tw (target options)

These options are called target options. You use them (with **c**, **e**, or **d**) to produce a .COM, .EXE, or .DLL file.

- **/Td** creates a DOS .EXE file.
- **/Tdc** creates a DOS .COM file.
- **/Tde** creates a DOS .EXE file.
- **/Tw** tells TLINK to create a Windows executable file. This option is not necessary if you include a module definition file with an EXETYPE Windows statement. With or without the **/Tw** option, if the included module definition file has a NAME statement, TLINK creates an application (.EXE); if the module definition file has a LIBRARY statement, TLINK creates a DLL.

  If no module definition file is included in the link, you must specify the **/Tw** or **/Twe** option for a Windows .EXE, or the **/Twd** option for a Windows DLL.

  None of the **/Tw** options are compatible with the **/o** option (overlay modules).
- **/Twe** creates Windows .EXE files. The **/Twe** option overrides a LIBRARY statement in the module definition file (which normally causes TLINK to create a DLL).
- **/Twd** creates Windows DLLs. The **/Twd** option overrides a NAME statement in the module definition file (which normally causes TLINK to create an .EXE file).

/v (debugging information)

The **/v** option directs TLINK to include debugging information in the executable file. If this option is found anywhere on the command line, debugging information will be included executable for all object modules that contained debugging information. You can use the **/v+** and **/v–** options to selectively enable or disable inclusion of debugging information on a module-by-module basis (but not on the same command line as **/v**). For example, this command

```
tlink mod1 /v+ mod2 mod3 /v- mod4
```

includes debugging information for modules *mod2* and *mod3*, but not for *mod1* and *mod4*.

TLINK can't generate debugging information for a .COM file (one created with the **/t** or **/Tdc** options). If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **–c** option; this creates a .COM file from an .EXE.

/y (expanded or extended memory)

This option controls TLINK's use of expanded or extended memory for I/O buffering. If, while reading object files or while writing the executable file, TLINK needs more memory for active data structures, it will either purge buffers or swap them to expanded or extended memory.

In the case of input file buffering, purging simply means throwing away the input buffer so that its space can be used for other data structures. In the case of output file buffering, purging means writing the buffer to its correct place in the executable file. In either case, you can substantially increase the speed of a link by allowing these buffers to be swapped to expanded or extended memory.

TLINK's capacity is not increased by swapping; only its performance is improved. By default, swapping to expanded memory is enabled, while swapping to extended memory is disabled. If swapping is enabled and no appropriate memory exists in which to swap, then swapping does not occur. If you link with TLINKX, the protected mode linker, neither of these options has an effect.

This option has several forms, shown below

/ye or **/ye+**	enable expanded memory swapping (default)
/ye-	disable expanded memory swapping
/yx or **/yx+**	enable extended memory swapping
/yx-	disable extended memory swapping (default)

Restrictions

Previous restrictions that no longer apply:

- TLINK now generates Windows .EXE and .DLL files.
- Common variables are now supported.
- Segments that are of the same name and class that are uncombinable are now accepted. They aren't combined, and they appear separately in the map file.
- Any Microsoft code can now be linked with TLINK.

TLINK can of course be used with Borland C++ (both the IDE and command-line versions), TASM, and other compilers.

The module definition file

The module definition file provides information to the linker about the contents and system requirements of a Windows application. More specifically, it

- names the application or dynamic link library (DLL)
- identifies the type of application as Windows or OS/2
- lists imported functions and exported functions
- describes the code and data segment attributes; allows you to specify attributes for additional code and data segments
- specifies the size of the heap and stack
- provides for the inclusion of a DOS stub program

Note that the IMPLIB utility can use a module definition file to create an import library (see page 192). The IMPDEF utility can actually create a module definition file for use with IMPLIB (see page 190).

Module definition file defaults

The module definition file is not strictly necessary to produce a Windows executable under Borland C++.

If no module definition file is specified, the following defaults are assumed.

CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE (for applications) or PRELOAD MOVEABLE SINGLE (for DLLs)
HEAPSIZE	4096
STACKSIZE	5120

To replace the EXETYPE statement, the Borland C++ linker can discover what kind of executable you want to produce by checking settings in the IDE or options on the command line.

You can include an import library to substitute for the IMPORTS section of the module definition.

You can use the **_export** keyword in the definitions of export functions in your C and C++ source code to remove the need for an EXPORTS section. Note, however, that if **_export** is used to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need to have a module definition file.

A quick example

Here's a module definition from the WHELLO example, discussed on page 104:

```
NAME           WHELLO
DESCRIPTION    'C++ Windows Hello World'
EXETYPE        WINDOWS
CODE           MOVEABLE
DATA           MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      5120
EXPORTS        MainWindowProc
```

Let's take this file apart, statement by statement:

- NAME specifies a name for an application. If you want to build a DLL instead of an application, you would use the LIBRARY statement instead. Every module definition file should have either a NAME statement or a LIBRARY statement, but never both. The name specified must be the same name as the executable file.
- DESCRIPTION lets you specify a string that describes your application or library.
- EXETYPE can be either WINDOWS or OS2. Only WINDOWS is supported in this version of Borland C++.
- CODE defines the default attributes of code segments. The MOVEABLE option means that the code segment can be moved in memory at run-time.
- DATA defines the default attributes of data segments. MOVEABLE means that it can be moved in memory at run-time. Windows lets you run more than one instance of an application at the same time. In support of that, the MULTIPLE options ensures that each instance of the application has its own data segment.
- HEAPSIZE specifies the size of the application's local heap.
- STACKSIZE specifies the size of the application's local stack. You can't use the STACKSIZE statement to create a stack for a DLL.

- EXPORTS lists those functions in the WHELLO application that will be called by other applications or by Windows. Functions that are intended to be called by other modules are called callbacks, callback functions, or export functions.
- To help you avoid the necessity of creating and maintaining long EXPORTS sections, Borland C++ provides the **_export** keyword. Functions flagged with **_export** will be identified by the linker and entered into an export table for the module. If the Smart Callbacks option is used at compile time (**/WS** on the BCC command-line, or **O**ptions | **C**ompiler | **E**ntry/Exit Code | Windows Smart Callbacks), then callback functions do *not* need to be listed either in the EXPORTS statement or flagged with the **_export** keyword. Borland C++ compiles them in such a way so that they can be callback functions.

This application doesn't have an IMPORTS statement, because the only functions it calls from other modules are those from the Windows API; those functions are imported via the automatic inclusion of the IMPORT.LIB import library. When an application needs to call other external functions, these functions must be listed in the IMPORTS statement, or included via an import library (see page 192 for a discussion of import libraries).

This application doesn't include a STUB statement. Borland C++ uses a built-in stub for Windows applications. The built-in stub simply checks to see if the application was loaded under Windows, and, if not, terminates the application with a message that Windows is required. If you want to write and include a custom stub, specify the name of that stub with the STUB statement.

Module definition reference

This section describes each statement in a module definition file.

CODE

The CODE statement defines the default attributes of code segments. Code segments can have any name, but must belong to segment classes whose name ends in CODE. For instance, valid segment class names are CODE or MYCODE. The syntax is

```
CODE [FIXED | MOVEABLE]
   [DISCARDABLE | NONDISCARDABLE]
   [PRELOAD | LOADONCALL]
```

FIXED means that the segment remains at a fixed memory location; MOVEABLE means that the segment can be moved.

DISCARDABLE means that the segment can be discarded if it is no longer needed. DISCARDABLE implies MOVEABLE. NONDISCARDABLE means that the segment can't be discarded.

PRELOAD means that the segment is loaded when the module is first loaded; LOADONCALL means that the segment is loaded when code in this segment is called. The Resource Compiler and the Windows loader set the code segment containing the initial program entry point to PRELOAD regardless of the setting in the module definition file.

Default attributes for code segments are

FIXED NONDISCARDABLE LOADONCALL

DATA

The DATA statement defines the default attributes of data segments.

The syntax of the DATA statement is

DATA [NONE | SINGLE | MULTIPLE] [FIXED | MOVEABLE]

NONE means that there is no data segment. If you specify NONE, do not include any other options. This option is available only for libraries.

SINGLE means that a single segment is shared by all instances of the module. MULTIPLE means that each instance of an application has a segment. SINGLE is only valid for libraries; MULTIPLE is only valid for applications.

FIXED means that the segment remains at a fixed memory location. MOVEABLE means that the segment can be moved.

The default attributes for data segments in applications are FIXED MULTIPLE. For libraries the default attributes are FIXED SINGLE.

The automatic data segment is the segment whose group is DGROUP. This physical segment also contains the local heap and stack (see the HEAPSIZE and STACKSIZE module definition file statements). The Resource Compiler and the Windows loader set the automatic data segment to be PRELOAD, regardless of the setting in the module definition file.

DESCRIPTION

The DESCRIPTION statement inserts text into the application module. The DESCRIPTION statement is typically used to embed author, date, or copyright information. DESCRIPTION is an optional statement. The syntax is

```
DESCRIPTION 'Text'
```

Text specifies an ASCII string delimited with single quotes.

EXETYPE

The EXETYPE statement specifies the default executable file (.EXE) header type (Windows or OS/2). You can only specify WINDOWS in this version of Borland C++. The syntax is

```
EXETYPE Type
```

Type determines the type of header TLINK writes to the executable file.

EXPORTS

The EXPORTS statement defines the names and attributes of the functions to be exported. The EXPORTS keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line. The syntax is

```
EXPORTS
   ExportName [Ordinal] [RESIDENTNAME] [NODATA] [Parameter]
```

ExportName specifies an ASCII string that defines the symbol to be exported. It has the following form:

```
<EntryName>=[InternalName]
```

InternalName is the name used within the application to refer to this entry. *EntryName* is the name listed in the executable file's entry table is externally visible.

Ordinal defines the function's ordinal value. It has the following form:

```
@ordinal
```

where *ordinal* is an integer value that specifies the function's ordinal value.

When an application module or DLL module calls a function exported from a DLL, the calling module can refer to the function by name or by ordinal value. In terms of speed, referring to the function by ordinal is faster since string comparisons are not required to locate the function. In terms of memory allocation, exporting a function by ordinal (from the point of view of that function's DLL) and importing/calling a function by ordinal (from the point of view of the calling module) is more efficient. When a function is exported by ordinal, the name resides in the non-resident name table. When a function is exported by name, the name resides in the resident name table. The resident name table for a module is resident in memory whenever the module is loaded; the non-resident name table isn't.

The RESIDENTNAME option lets you specify that the function's name must be resident at all times. This is useful only when exporting by ordinal (when the name wouldn't be resident by default).

The NODATA option lets you specify that the function is not bound to a specific data segment. The function will use the current data segment.

Parameter is an optional integer value that specifies the number of words the function expects to be passed as parameters.

HEAPSIZE

The HEAPSIZE statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory. The syntax is

HEAPSIZE *bytes*

bytes is an integer value that specifies the heap size in bytes. It must not exceed 65,536 (the physical segment size).

The default heap size is zero. The minimum size is 256 bytes. The sum total of the automatic data segment (DGROUP), the local heap, and the stack must not exceed 65,536.

IMPORTS

The IMPORTS statement defines the names and attributes of the functions to be imported from dynamic link libraries. Instead of listing imported DLL functions in the IMPORTS statement, you can either specify an import library for the DLL in the TLINK command line, or—in the IDE—include the import library for the DLL in the project.

The IMPORTS key word marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line. The syntax is

IMPORTS
  [*InternalName*=]*ModuleName.Entry*

InternalName is an ASCII string that specifies the unique name that the application will use to call the function.

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module that contains the function. The module name must match the name of the executable file. For example, the file SAMPLE.DLL has the the module name SAMPLE.

Entry specifies the function to be imported. It can be either an ASCII string that names the function, or an integer that gives the function's ordinal value.

LIBRARY

The LIBRARY statement defines the name of a DLL module. A module definition file can contain either a NAME statement to indicate an application or a LIBRARY statement to indicate a DLL, but not both.

Like an application's module name, a library's module name must match the name of the executable file. For example, the library MYLIB.DLL has the module name MYLIB. The syntax is

LIBRARY *LibraryName*

LibraryName specifies an ASCII string that defines the name of the library module.

The start address of the library module is determined by the library's object files; it is an internally defined function.

LibraryName is optional. If the parameter is not included, TLINK uses the filename part of the executable file (that is, the name with the extension removed).

If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

NAME

The NAME statement defines the name of the application's executable module. The module name identifies the module when exporting functions. The syntax is

NAME *ModuleName*

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name "SAMPLE".

The *ModuleName* parameter is optional. If the parameter is not included, TLINK assumes that the module name matches the filename of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, TLINK assumes that the module name is "MYAPP"

If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

SEGMENTS

The SEGMENTS statement defines the segment attributes of additional code and data segments. The syntax is

```
SEGMENTS
   SegmentName [CLASS 'ClassName'] [MinAlloc]
   [FIXED | MOVEABLE] [DISCARDABLE |
   NONDISCARDABLE]
   [SHARED | NONSHARED] [PRELOAD | LOADONCALL]
```

SegmentName specifies a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.

ClassName is an optional key word that specifies the class name of the specified segment. If no class name is specified, TLINK uses the class name CODE by default.

MinAlloc is an optional integer value that specifies the minimum allocation size for the segment. Currently, TLINK ignores this value.

FIXED means that the segment remains at a fixed memory location. The MOVEABLE option means that the segment can be moved if necessary, in order to compact memory.

DISCARDABLE means that the segment can be discarded if it is no longer needed; NONDISCARDABLE means that the segment can not be discarded.

PRELOAD means that the segment is loaded immediately; LOADONCALL means that the segment is loaded when it is accessed or called. The Resource Compiler may override the LOADONCALL option and preload segments instead.

Default attributes for additional segments are as described for CODE and DATA segments (depending on the type of additional segment).

STACKSIZE The STACKSIZE statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it makes function calls. Do not use the STACKSIZE statement for dynamic link libraries. The syntax is

```
STACKSIZE bytes
```

bytes is an integer value that specifies the stack size in bytes.

If the application makes no function calls, STACKSIZE defaults to 0. If your application does make function calls the minimum size is 5120 bytes (if you specify less, it will be changed to 5120). The sum total of the automatic data segment (DGROUP), the local heap, and the stack must not exceed 65,536.

STUB The STUB statement appends a DOS executable file specified by *FileName* to the beginning of the module. The executable stub should display a warning message and terminate if the user doesn't have Windows loaded.

Borland C++ adds a built-in stub to the beginning of a Windows application unless a different stub is specified with the STUB statement. Therefore, you should not use the STUB statement merely to include WINSTUB.EXE, because the linker will do this for you automatically.

The syntax is

```
STUB "FileName"
```

FileName specifies the name of the DOS executable file that will be appended to the module. The name must have the DOS file name format.

If the file named by *FileName* is not in the current directory, TLINK searches for the file in the directories specified by the user's PATH environment variable.

TLINK messages

TLINK has three types of messages: fatal errors, errors, and warnings.

- A fatal error causes TLINK to stop immediately; the .EXE file is deleted.
- An error (also called a nonfatal error) does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file. Errors are treated as fatal errors in the IDE.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

In manual	What you'll see onscreen
errorcode	Error code number for internal errors
filename	A file name (with or without extension)
group	A group name
linenum	The line number within a file
module	A module name
segment	A segment name
symbol	A symbol name
XXXXh	A 4-digit hexadecimal number, followed by *h*

Messages are listed in ASCII alphabetic order; messages beginning with variable names or numbers come first. Since messages that begin with one of the variables just listed cannot be alphabetized by what you will actually see when you receive such a message, all such messages have been placed at the beginning of the message list.

If the variable occurs later in the text of the error message (for example, "Invalid segment definition in module *module*"), you can find the message in correct alphabetical order; in this case, under *I*.

Warning

***filename* (*linenum*): Duplicate external name in exports**

Two export functions listed in the EXPORTS section of a module definition file defined the same external name. For instance,

```
EXPORTS
   AnyProc=MyProc1
   AnyProc=MyProc2
```

Warning ***filename*** **(*linenum*): Duplicate internal name in exports**
Two export functions listed in the EXPORTS section of the module definition file defined the same internal name. For example,

```
EXPORTS
   AnyProc1=MyProc
   AnyProc2=MyProc
```

Warning ***filename*** **(*linenum*): Duplicate internal name in imports**
Two import functions listed in the IMPORTS section of the module definition file defined the same internal name. For instance,

```
IMPORTS
   AnyProc=MyMod1.MyProc1
   AnyProc=MyMod2.MyProc2
```

or

```
IMPORTS
   MyMod1.MyProc
   MyMod2.MyProc
```

Fatal error ***filename*** **(*linenum*): File read error**
A DOS error occurred while TLINK read the module definition file. This usually means that a premature end of file occurred.

Fatal error ***filename*** **(*linenum*): Incompatible attribute**
This error indicates that TLINK encountered incompatible segment attributes in a CODE or DATA statement (for instance, both PRELOAD and LOADONCALL can't be attributes for the same segment).

Fatal error ***filename*** **(*linenum*): Missing internal name**
In the IMPORTS section of the module definition file there was a reference to an entry specified via module name and ordinal number. When an entry is specified by ordinal number an internal name must be assigned to this import definition. It is this internal name that your program uses to refer to the imported definition. The syntax in the module definition file should be:

```
<internalname>=<modulename>.<ordinal>
```

Fatal error ***filename*** **(*linenum*): Syntax error**
TLINK found a syntax error in the module definition file. The filename and line number tell you where the syntax error occurred.

Warning **_symbol_ conflicts with module _module_ in module _module_**
This indicates an inconsistency in the definition of *symbol.* This either means that two virtual functions of this name were encountered with different sizes, or that TLINK found one virtual function and one command definition with the same name.

Error or Warning **_symbol_ is duplicated in module _module_**
This message can result from a conflict between two symbols (either public or communal) defined in the same module. An error occurs if both are encountered in an .OBJ file. A warning is issued if TLINK finds the duplicates in a library; in this case, TLINK uses the first definition.

Error or Warning **_symbol_ defined in module _module_ is duplicated in module _module_**
This message can result from a conflict between two symbols (either public or communal). This usually means that a symbol is defined in two modules. An error occurs if both are encountered in the .OBJ file(s), because TLINK doesn't know which is valid. A warning results if TLINK finds one of the duplicated symbols in a library and finds the other in an .OBJ file; in this case, TLINK uses the one in the .OBJ file.

Fatal error **32-bit record encountered**
This message occurs when an object file that contains 80386 32-bit records is encountered, and the **/3** option has not been used. Simply restart TLINK with the **/3** option.

Warning **Attempt to export non-public symbol _symbol_**
A symbol name was listed in the EXPORTS section of the module definition file, but no symbol of this name was found as public in the modules linked. This either implies a mistake in spelling or case, or that a procedure of this name was not defined.

Error **Automatic data segment exceeds 64K**
The sum of the DGROUP physical segment, local heap, and stack exceeded 64K. Either specify smaller values for the HEAPSIZE and STACKSIZE statements in the module definition file, or decrease the size of your near data in DGROUP. The map file will show the sizes of the component segments in DGROUP. The **/s** TLINK command-line option may be useful to help you find the module.

Fatal error **Bad character in parameters**
One of the following characters was encountered in the command line or in a response file:

```
" * < = > ? [ ] |
```

or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.

Fatal error **Bad object file *filename***
An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was pressed.

Fatal error **Bad version number in parameter block**
This error indicates an internal inconsistency in the IDE. If it occurs, exist and restart the IDE. This error will not occur in the standalone version.

Fatal error **Cannot generate COM file: data below initial CS:IP defined**
This error results from trying to generate data or code below the starting address (usually 100) of a .COM file. Be sure that the starting address is set to 100 by using the (ORG 100H) instruction. This error message should not occur for programs written in a high-level language. If it does, ensure that the correct startup (C0*x*) object module is being linked in.

Fatal error **Cannot generate COM file: invalid initial entry point address**
You used the **/Tdc** or **/t** option, but the program starting address is not equal to 100H, which is required with .COM files.

Fatal error **Cannot generate COM file: program exceeds 64K**
You used the **/Tdc** or **/t** option, but the total program size exceeds the .COM file limit.

Fatal error **Cannot generate COM file: segment-relocatable items present**
You used the **/Tdc** or **/t** option, but the program contains segment-relative fixups, which are not allowed with .COM files.

Fatal error **Cannot generate COM file: stack segment present**
You used the **/Tdc** or **/t** option, but the program declares a stack segment, which is not allowed with .COM files.

Error **Common segment exceeds 64K**
The program had more than 64K of near uninitialized data. Try declaring some uninitialized data as far.

Warning **Debug info switch ignored for .COM files**
Borland C++ does not include debug information for .COM files. See the description of the **/v** option on page 248.

Fatal error **DOS error, ax = *number***
This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes where this error could occur are read, write, seek, and close.

Warning **Duplicate ordinal *number* in exports**
This warning occurs when TLINK encounters two exports with the same ordinal value. First check the module definition file to ensure that there are no duplicate ordinal values specified in the EXPORTS section. If not, then you are linking with modules which specify exports by ordinals and one of two things happened: either two export records specify the same ordinal, or the exports section in the module definition file duplicates an ordinal in an export record.

Export records (EXPDEF) are comment records found in object files and libraries which specify that particular variables are to be exported. Optionally, these records can specify ordinal values when exporting by ordinal (rather than by name).

Error **Fixup overflow at *segment*:*xxxx*h, target = *segment*:*xxx*h in module *module***
Fixup overflow at *segment*:*xxxx*h, target = *symbol* in module *module*
Either of these messages indicate an incorrect data or code reference in an object file that TLINK must fix up at link time.

This message is most often caused by a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named module, so look in the source file of that module for the offending reference.

In an assembly language program, a fixup overflow frequently occurs if you have declared an external variable within a segment definition, but this variable actually exists in a different segment.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or a high-level language besides Borland C++, there may be other possible causes for this message. Even in Borland C++, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Fatal error **Group *group* exceeds 64K**
This message will occur if a group exceeds 64K bytes when the segments of the group are combined.

Warning **Group *group1* overlaps group *group2***
This means that TLINK has encountered nested groups. This warning only occurs when overlays are used or when linking a Windows program.

Fatal error **Illegal group definition: *group* in module *module***
This error results from a malformed GRPDEF record in an .OBJ file. This latter case could result from custom-built .OBJ files or a bug in the translator used to generate the .OBJ file. If this occurs in a file created by Borland C++, recompile the file. If the error persists, contact Borland.

Fatal error **Internal linker error *errorcode***
An error occurred in the internal logic of TLINK. This error shouldn't occur in practice, but is listed here for completeness in the event that a more specific error isn't generated. If this error persists, write down the *errorcode* number and contact Borland.

Fatal error **Internal undefined error**
An error occurred in the internal logic of TLINK. This error shouldn't occur in practice, but is listed here for completeness in the event that a more specific error isn't generated. If this error persists, contact Borland.

Warning **Invalid entry at *segment:xxxxh***
A rare internal error of TLINK, listed here for completeness. This error indicates that a necessary entry was missing from the entry table of a Windows executable file. If it persists, contact Borland.

Error **Invalid entry point offset**
This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.

Fatal error **Invalid initial stack offset**
This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.

Fatal error **Invalid limit specified for code segment packing**
This error occurs if you used the **/P** option and specified a size limit that was out of range. To be valid, the size limit must be between 1 and 65536 bytes; the default is 4096.

Fatal error **Invalid segment definition in module *module***
This message will generally occur only if a compiler produced a flawed object file. If this occurs in a file created by Borland C++, recompile the file. If the problem persists, contact Borland.

Error **Invalid size specified for segment alignment**
This error occurs if an invalid value is specified for the **/A** option. The size specified with **/A** must be an integral multiple of 2 and less than 64K. Common values are 16 and 512. This error only occurs when linking for Windows.

Warning **No automatic data segment**
No group named DGROUP was found. Because the Borland C++ initialization files define DGROUP, you will only see this error if you don't link with an initialization file and your program doesn't define DGROUP. Windows uses DGROUP to find the local data segment. The DGROUP is required for Windows applications (but not DLLs) unless DATA NONE is specified in the module definition file.

Warning **No module definition file specified: using defaults**
TLINK was invoked with one of the Windows options, but no module definition file was specified. See page 250 for more information about module definition file defaults.

Warning **No program starting address defined**
This warning means that no module defined the initial starting address of the program. This is almost certainly caused by forgetting to link in the initialization module C0*x*.OBJ. This warning should not occur when linking a Windows DLL.

Warning **No stack**
This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Borland C++, or

for any application program that will be converted to a .COM file. For other programs (except DLLs), this indicates an error.

If a Borland C++ program produces this message for any but the tiny memory model, make sure you are using the correct C0*x* startup object files.

Warning **No stub for fixup at *segment:xxxx*h in module *module***
This error occurs when the target for a fixup is in an overlay segment, but no stub is found for a target external. This is usually the result of not making public a symbol in an overlay that is referenced from the same module.

Fatal error **Not enough memory**
There is not enough memory to run TLINK. Try removing any TSR applications currently loaded, or reduce the size of any RAM disk currently active. Then run TLINK again.

Fatal error **Out of memory**
TLINK has run out of dynamically allocated memory needed during the link process. This error is a catchall for running into a TLINK limit on memory usage. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together.

Warning **Overlays generated and no overlay manager included**
This warning is issued if overlays are created but the symbol __OVRTRAP__ is not defined in any of the object modules or libraries linked in. The standard overlay library (OVERLAY.LIB) defines this symbol.

Warning **Overlays ignored in new executable image**
This error occurs if you attempt to link a Windows program with the **/o** option on. Windows executables can't be overlaid, although, with discardable code segments, you should be able to achieve a similar effect.

Error **Program entry point may not reside in an overlay**
Although almost all of an application can be overlaid, the initial starting address cannot reside in an overlay. This error usually means that an attempt was made to overlay the initialization module C0*x*.OBJ, for instance, by specifying the **/o** option before the startup module.

Fatal error **Relocation item exceeds 1MB DOS limit**
The DOS executable file format doesn't support relocation items for locations exceeding 1MB. Although DOS could never *load* an image this big, DOS extenders can, and thus TLINK supports generating images greater than DOS could load. Even if the image is loaded with a DOS extender, the DOS executable file format is limited to describing relocation items in the first 1MB of the image.

Fatal error **Relocation offset overflow**
This error only occurs for 32-bit object modules and indicates a relocation (segment fixup) offset greater than the DOS limit of 64K.

Fatal error **Relocation table overflow**
This error only occurs for 32 bit object modules. The file being linked contains more base fixups than the standard DOS relocation table can hold (base fixups are created mostly by calls to far functions).

Fatal error **Segment *segment* exceeds 64K**
This message occurs if too much data is defined for a given data or code segment when TLINK combines segments with the same name from different source files.

Warning **Segment *segment* is in two groups: *group1* and *group2***
The linker found conflicting claims by the two named groups. Usually, this only happens in assembly language programs. It means that two modules assigned the segment to two different groups.

Fatal error **Segment alignment factor too small**
This error occurs if the segment alignment factor (set with the **/A** option) is too small to represent the file addresses of the segments in the .EXE file. This error only occurs when linking for Windows. See the documentation for the **/A** option on page 242 for more information.

Fatal error **Segment too large for segment table**
This error should never occur in practice. It means that a segment was bigger than 64K and its size cannot be represented in the executable file. This error can only occur when linking for Windows; the format of the executable file used for Windows does not support segments greater than 64K.

Fatal error **Stub program exceeds 64K**
This error occurs if a DOS stub program written for a Windows application exceeds 64K. Stub programs are specified via the STUB module definition file statement. TLINK only supports stub programs up to 64K.

Fatal error **Table limit exceeded**
This message results from one of linker's internal tables overflowing. This usually means that the programs being linked have exceeded the linker's capacity for public symbols, external symbols, or for logical segment definitions. Each instance of a distinct segment name in an object file counts as a logical segment; if two object files define this segment, then this results in two logical segments

Error **Too many error or warning messages**
On the command line or in the IDE, you can set a limit on the number of errors or warnings that can occur before linking is stopped. This error indicates that TLINK reached that limit.

Fatal error **Unable to open file *filename***
This occurs if the named file does not exist or is misspelled.

Error **Undefined symbol *symbol* in module *module***
The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check to make sure the symbol is spelled correctly.

You will usually see this error from TLINK for Borland C++ symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs. If you are linking C++ code with C modules, you might have forgotten to wrap C external declarations in *extern "C" {...}*.

Fatal error **Unknown option**
A forward slash character (/), hyphen (–), or DOS switch character was encountered on the command line or in a response file without being followed by one of the allowed options. This might mean that you used the wrong case to specify an option.

Error **User break**
TLINK aborts linking when the *Ctrl-Break* key is pressed.

Fatal error **Write failed, disk full?**
This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.

A P P E N D I X

A

Precompiled headers

Borland C++ can generate and subsequently use precompiled headers for your projects. Precompiled headers can greatly speed up compilation times.

How they work

When compiling large C and C++ programs, the compiler can spend up to half of its time parsing header files. When the compiler parses a header file, it enters declarations and definitions into its symbol table. If ten of your source files include the same header file, this header file is parsed ten times, producing the same symbol table every time.

Precompiled header files cut this process short. During one compilation, the compiler stores an image of the symbol table on disk in a file called TCDEF.SYM by default. (TCDEF.SYM is stored in the same directory as the compiler.)Later, when the same source file is compiled again (or another source file that includes the same header files), the compiler reloads TCDEF.SYM from disk instead of parsing all the header files again. Directly loading the symbol table from disk is over ten times faster than parsing the text of the header files.

Precompiled headers will only be used if the second compilation uses one or more of the same header files as the first one, and if a lot of other things, like compiler options, defined macros and so on, are also identical.

If, while compiling a source file, Borland C++ discovers that the first **#include**s are identical to those of a previous compilation (of the same source or a different source), it will load the binary image for those **#include**s, and parse the remaining **#include**s.

Use of precompiled headers for a given module is an all or nothing deal: the precompiled header file is not updated for that module if compilation of any included header file fails.

Drawbacks

When using using precompiled headers, TCDEF.SYM can become very big, because it contains symbol table images for all sets of includes encountered in your sources. You can reduce the size of this file; see "Optimizations" on page 274.

If a header contains any code, then it can't be precompiled. For example, while C++ class definitions may appear in header files, you should take care that only member functions that are inline are defined in the header; heed warnings such as "Functions containing for are not expanded inline".

Using precompiled headers

You can control the use of precompiled headers in any of three ways:

- from within the IDE, using the **O**ptions | **C**ompiler | **C**ode Generation dialog box (see page 62). The IDE bases the name of the precompiled header file on the project name, creating *PROJECT*.SYM
- from the command line using the **–H**, **–H=*filename***, and **–Hu** options (see page 178)
- or from within your code using the pragmas **hdrfile** and **hdrstop** (see Chapter 4 in the *Programmer's Guide*)

Setting file names

The compiler uses just one file to store all precompiled headers. The default file name is TCDEF.SYM. You can explicitly set the name with the **–H=*filename*** command-line option or the `#pragma hdrfile` directive.

Caution! You may notice that your .SYM file is smaller than it should be. If this happens, the compiler may have run out of disk space when writing to the .SYM file. When this happens, the compiler deletes the .SYM in order to make room for the .OBJ file, then starts creating a new (and therefore shorter) .SYM file. If this happens, just free up some disk space before compiling.

Establishing identity

The following conditions need to be identical in order for a previously generated precompiled header to be loaded for a subsequent compilation.

The second or later source file must:

- have the same set of include files in the same order
- have the same macros defined to identical values
- use the same language (C or C++)
- use header files with identical time stamps; these header files can be included either directly or indirectly

In addition, the subsequent source file must be compiled with the same settings for the following options:

- memory model, including SS != DS (**-m*x***)
- underscores on externs (**-u**)
- maximum identifier length (**-iL**)
- target DOS (default) or Windows (**-W** or **–W*x***)
- generate word alignment (**-a**)
- Pascal calls (**-p**)
- treat enums as integers (**-b**)
- default char is unsigned (**-K**)
- virtual table control (**-V*x***)

Optimizing precompiled headers

For Borland C++ to most efficiently compile using precompiled headers, follow these rules:

- Arrange your header files in the same sequence in all source files.
- Put the largest header files first.
- Prime TCDEF.SYM with often-used initial sequences of header files.
- Use **#pragma hdrstop** to terminate the list of header files at well-chosen places. This lets you make the list of header files in different sources look similar to the compiler. **#pragma hdrstop** is described in more detail in Chapter 4 in the *Programmer's Guide.*

For example, given the two source files ASOURCE.C and BSOURCE.C, both of which include windows.h and myhdr.h,

ASOURCE.C:

```
#include <windows.h>
#include "myhdr.h"
#include "xxx.h"
<...>
```

BSOURCE.C:

```
#include "zz.h"
#include <string.h>
#include "myhdr.h"
#include <windows.h>
<...>
```

You would rearrange the beginning of BSOURCE.C to:

Revised BSOURCE.C:

```
#include <windows.h>
#include "myhdr.h"
#include "zz.h"
#include <string.h>
<...>
```

Note that windows.h and myhdr.h are in the same order in BSOURCE.C as they are in ASOURCE.C. You could also make a new source called PREFIX.C containing only the header files, like this:

PREFIX.C

```
#include <windows.h>
#include "myhdr.h"
```

If you compile PREFIX.C first (or insert a `#pragma hdrstop` in both ASOURCE.C and BSOURCE.C after the `#include "myhdr.h"` statement) the net effect is that after the initial compilation of PREFIX.C, both ASOURCE.C and BSOURCE.C will be able to load the symbol table produced by PREFIX.C. The compiler will then only need to parse xxx.h for ASOURCE.C and zz.h and string.h for BSOURCE.C.

I N D E X

2.0

USER'S
GUIDE

BORLAND® C++

B O R L A N D

CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300.
OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, SWEDEN AND THE UNITED KINGDOM ■ PART # 14MN-TCP01 ■ BOR 1973