Turbo C®++

User’s Guide
INTRODUCTION
Typefaces used in these books 2

CHAPTER 1 THE IDE REFERENCE 5
Part 1: Starting up and exiting 6
Command-line options 6
The /b option 6
The /m option 6
The /d option 7
The /e and /x options 7
The /rx option 7
The /l option 8
The /p option 8
Exiting Turbo C++ 8
Part 2: The components 8
The menu bar and menus 9
Shortcuts 10
Full Menus On and Off 13
Turbo C++ windows 14
Window management 16
The status line 17
Dialog boxes 18
Check boxes and radio buttons 19
Input boxes and lists 19
Editing 20
Part 3: Menu reference 21
≡ (System) menu 21
About 21
Clear Desktop 21
Repaint Desktop 22
Transfer items 22
File menu 22
Open 22
New 24
Save 24
Save As 24
Save All 24
Change Dir 25
Print 25
Get Info 26
DOS Shell 26
Quit 27
Edit menu 27
Restore Line . 28
Cut 28
Copy 28
Paste 28
Copy Example 28
Show Clipboard 29
Clear 29
Search menu 29
Find 29
Replace 31
Search Again 32
Go to Line Number 32
Previous Error 33
Next Error 33
Locate Function 33
Run menu 33
Run 34
Program Reset 35
Go to Cursor 35
Trace Into 35
Step Over 36
Arguments 37
Compile menu 37
Compile to OBJ 38
Make EXE File 38
Link EXE File 38
Build All 38
Remove Messages 38
Debug menu 39
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Manipulating windows</td>
<td>16</td>
</tr>
<tr>
<td>1.2</td>
<td>Get Info settings</td>
<td>26</td>
</tr>
<tr>
<td>1.3</td>
<td>Format specifiers recognized in debugger expressions</td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Full summary of editor commands</td>
<td>102</td>
</tr>
<tr>
<td>3.2</td>
<td>Block commands in depth</td>
<td>105</td>
</tr>
<tr>
<td>3.3</td>
<td>Other editor commands in depth</td>
<td>106</td>
</tr>
<tr>
<td>3.4</td>
<td>Delimiter pairs</td>
<td>110</td>
</tr>
<tr>
<td>4.1</td>
<td>Command-line options summary</td>
<td>114</td>
</tr>
<tr>
<td>5.1</td>
<td>MAKE directives</td>
<td>159</td>
</tr>
<tr>
<td>6.1</td>
<td>Editor key commands</td>
<td>208</td>
</tr>
<tr>
<td>6.2</td>
<td>Customizing keystrokes</td>
<td>208</td>
</tr>
</tbody>
</table>
1.1: A typical window .................. 15
1.2: A typical status line ............... 17
1.3: A typical dialog box ................ 18
1.4: The Load a File dialog box ......... 22
1.5: The Save File As dialog box ........ 24
1.6: The Change Dir dialog box .......... 25
1.7: The Get Info box .................... 26
1.8: The Find dialog box .................. 30
1.9: The Replace dialog box .............. 32
1.10: The Go to Line Number dialog box .. 33
1.11: The Locate Function dialog box ... 33
1.12: The Arguments dialog box .......... 37
1.13: The Evaluate/Modify dialog box ... 43
1.14: The Breakpoints dialog box ......... 47
1.15: The Breakpoint Modify/New dialog box .................. 48
1.16: The Project File dialog box ......... 49
1.17: The Add Item to Project List dialog box .................. 50
1.18: The Override Options dialog box ... 51
1.19: The Include Files dialog box ........ 52
1.20: The Code Generation dialog box ... 53
1.21: The Advanced Code Generation dialog box .................. 55
1.22: The Optimizations Options dialog box .................. 58
1.23: The Source Options dialog box .... 59
1.24: The Compiler Messages dialog box .. 60
1.25: The Portability dialog box ......... 61
1.26: The ANSI Violations dialog box ... 61
1.27: The More ANSI Violations dialog box .................. 62
1.28: The C++ Warnings dialog box .... 62
1.29: The Frequent Errors dialog box ... 63
1.30: The More Frequent Errors dialog box .................. 63
1.31: The Segment Names dialog box .... 64
1.32: The Transfer dialog box .......... 64
1.33: The Modify/New Transfer Item dialog box .................. 65
1.34: The Make dialog box ............... 70
1.35: The Linker dialog box .............. 71
1.36: The Debugger dialog box .......... 72
1.37: The Directories dialog box ......... 74
1.38: The Preferences dialog box ........ 75
3.1: Search for match to square bracket or parenthesis .................. 110
3.2: Forward search I .................... 111
3.3: Forward search II .................... 111
3.4: Backward search ..................... 112
5.1: Detailed map of segments ........... 183
If you haven’t done so already, read the introduction, Chapter 1 (“Installing Turbo C++”), and Chapter 2 (“Navigating the Turbo C++ manuals”) in Getting Started for information on the overall organization of the Turbo C++ manuals. Those chapters tell you about many of the highlights of Turbo C++, how to install Turbo C++, and how to use the manuals most effectively.

This book, the User’s Guide, contains reference-style information on the integrated environment, the Project Manager, Turbo C++’s editor, the command-line compiler, utilities, and customization. The Programmer’s Guide provides useful material for the experienced C user (a language reference, C++ streams, memory models, mixed-model programming, video functions, floating point issues, overlays, error messages, and so on). The Library Reference contains a detailed list and explanation of Turbo C++’s extensive library functions and global variables.

Here is a breakdown of the chapters in this book:

**Chapter 1: The IDE reference** provides a complete reference to the menu system.

**Chapter 2: Managing multi-file projects** tells how to use the Project Manager to manage multi-file projects.

**Chapter 3: The editor from A to Z** provides a complete reference to the editor.

**Chapter 4: The command-line compiler** tells how to use the command-line compiler. It also explains configuration files.

**Chapter 5: Utilities** describes some of the utility programs that come with Turbo C++.

**Chapter 6: Customizing Turbo C++** tells how to adjust onscreen colors, editor defaults, compiler and linker defaults, and many other aspects of Turbo C++ with TCINST.
Appendix A: Turbo Editor macros describes the Turbo Editor Macro Language, a powerful utility you can use to enhance or change the Turbo C++ editor.

Typefaces used in these books

All typefaces used in this manual were produced by Borland's Sprint: The Professional Word Processor, on a PostScript laser printer. Their uses are as follows:

Monospace type This typeface represents text as it appears onscreen or in a program, or anything you must type (such as TC to start up Turbo C++).

ALL CAPS We use all capital letters for the names of constants and files, except for header files, which are traditionally represented in all lowercase letters.

[] Square brackets in text or DOS command lines enclose optional input or data that depends on your system. Text of this sort should not be typed verbatim.

<> Angle brackets in the function reference section enclose the names of include files.

Boldface Turbo C++ function names (such as printf) and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used, in text but not in program examples, for Turbo C++ reserved words (such as char, switch, near, and cdecl), for format specifiers and escape sequences (%d, \t), and for command-line options (/A).

Italics Italics indicate variable names (identifiers) that appear in text. They can represent terms that you can use as-is, or that you can think up new names for (your choice, usually). They are also used to emphasize certain words (especially new terms).
Keycaps

This typeface indicates a key on your keyboard. It is often used to describe a particular key you should press; for example, “Press Esc to exit a menu.”

This icon indicates keyboard actions.

This icon indicates mouse actions.
The IDE reference

Turbo C++ makes it easy and efficient for you to program. Everything you need to write, edit, compile, link, and debug your programs is at your fingertips when you start Turbo C++. That's what an integrated development environment (IDE) is all about.

The Turbo C++ environment also furnishes these extras to make program writing even smoother:

- multiple, movable, resizable windows
- mouse support
- dialog boxes
- cut-and-paste commands (with copying allowed from the Help window and between Edit windows)
- quick transfer to other programs (like TASM) and back again
- editor macro language

To smooth your introduction to the new IDE, we've divided this chapter into three parts: Part 1 tells you how to enter and exit the IDE; Part 2 discusses the generic components that comprise the IDE; and Part 3 provides information on the individual menu items, dialog boxes, buttons, and so on.

Part 1: Starting up and exiting

Starting up Turbo C++ is simple. You just move to your Turbo C++ directory and type `tc` at the DOS command line. If you like,
you can use one or more options along with the `TC` command. These options control automatic builds and makes and use of dual monitors, expanded and extended memory, RAM disks, LCD screens, and the EGA palette.

Command-line options

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

\[
\text{TC [sourcename | projectname] [option [option...]]}
\]

where `sourcename` is any ASCII file, `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 Turbo C++ to recompile and link all the files in your project, print the compiler messages to the standard output device, and then return to DOS. This option allows you to invoke Turbo C++ from a batch file so you can automate builds of projects. Before the build, Turbo C++ will load a default project file or one given on the command line. Turbo 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 `tc` command with either `/b` alone or the project file name followed by `/b`:

```
tc /b
```

```
tc 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 `tc` command, followed by `/b`:

```
tc myprog /b
```

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 /d option

The /d option causes Turbo 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 Turbo C++ screen will appear on the inactive monitor, and program output will go to the active monitor. So when you type tc /d at the DOS prompt on one monitor, Turbo C++ will come up on the other monitor. When you want to test your program on a particular monitor, exit Turbo C++, switch the active monitor to the one you want to test with, and then issue the tc /d command again. Program output will then go to the monitor where you typed the tc 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 Turbo C++ dual monitor option (/d).

The /e and /x options

Normally, Turbo C swaps to a hard disk when allocating memory. If you have expanded or extended memory, use the /e or /x options respectively to improve performance.

The /rx 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.
The \textit{/I} option

Use the \textit{/I} option if you’re running Turbo C on an LCD screen.

The \textit{/p} option

Use the \textit{/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.

Exiting Turbo C++

There are three ways to leave Turbo C++. The first method exits Turbo C++ “permanently,” you have to type \texttt{TC} again to reenter Turbo C++. The other two methods let you leave Turbo C++ to either type commands on the DOS command line, or to transfer temporarily to another program. Both of those methods then return you to Turbo C++.

- To exit Turbo C++ “permanently,” choose \texttt{File} | \texttt{Quit} (or press \texttt{Alt-X}). If you’ve made changes that you haven’t saved, Turbo C++ gives you a prompt asking if you want to save your programs before exiting.

- To leave Turbo C++ to enter commands at the DOS command line, choose \texttt{File} | \texttt{DOS Shell}. Turbo C++ stays in memory, but you’re transferred to DOS. 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 Turbo C++, type \texttt{EXIT} at the command line and press \texttt{Enter}. Turbo C++ reappears just as you left it.

- To temporarily transfer to another program without leaving Turbo C++, choose a program from the \texttt{=} menu. If there are no programs installed on this menu, you can add some with the \texttt{Options} | \texttt{Transfer} command.

Part 2: 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
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.

If a menu command is followed by an ellipsis mark (…), 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). A command without either an ellipsis mark or an arrow indicates that once you choose it, that action occurs.

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

1. Press F10. This makes the menu bar active, which means the next thing you type pertains to it, and not to any other IDE component.

   You’ll see a highlighted menu title when the menu bar is active. The menu title that’s highlighted is the currently selected menu.

2. Use the arrow keys to select the menu you want to display. Then press Enter.

   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 quickly display the Edit menu. From anywhere, press Alt and the highlighted letter 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, Turbo C++ either carries out the command, displays a dialog box, or displays another menu.

   You can also use a mouse to choose commands. The process is this:
1. Click the desired menu title to display the menu.
2. Click the desired command.

You can also 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. You can, however, still select (highlight) an unavailable command in order to get online help about it.

**Shortcuts**

Turbo C++ offers a number of quick ways to choose menu commands. For example, mouse users can combine the two-step process into one by dragging from the menu title down to the menu commands and releasing the mouse button when the command you want is selected.

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. (When moving from an input box to a group of buttons or boxes, you need to hold down *Alt* while pressing the highlighted letter.)

Here's a list of the shortcuts available:

<table>
<thead>
<tr>
<th>Do this...</th>
<th>To accomplish this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press <em>Alt</em> plus the highlighted letter of the command (just press the highlighted letter in a dialog box). For the <em>Edit</em> menu, press <em>Alt</em>-Spacebar.</td>
<td>Display the menu or carry out the command.</td>
</tr>
<tr>
<td>Type the keystrokes next to a menu command.</td>
<td>Carry out the command.</td>
</tr>
</tbody>
</table>

For example, to cut selected text, you can 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 Turbo C++ hot keys.
## General hot keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help</td>
<td>Displays a help screen.</td>
</tr>
<tr>
<td>F2</td>
<td>File</td>
<td>Save</td>
</tr>
<tr>
<td>F3</td>
<td>File</td>
<td>Open</td>
</tr>
<tr>
<td>F4</td>
<td>Run</td>
<td>Go to Cursor</td>
</tr>
<tr>
<td>F5</td>
<td>Window</td>
<td>Zoom</td>
</tr>
<tr>
<td>F6</td>
<td>Window</td>
<td>Next</td>
</tr>
<tr>
<td>F7</td>
<td>Run</td>
<td>Trace Into</td>
</tr>
<tr>
<td>F8</td>
<td>Run</td>
<td>Step Over</td>
</tr>
<tr>
<td>F9</td>
<td>Compile</td>
<td>Make EXE</td>
</tr>
<tr>
<td>F10</td>
<td>(none)</td>
<td>Takes you to the menu bar.</td>
</tr>
</tbody>
</table>

## Menu hot keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt-Spacebar</td>
<td>= menu</td>
<td>Takes you to the = (System)</td>
</tr>
<tr>
<td>Alt-C</td>
<td>Compile menu</td>
<td>Takes you to the Compile menu</td>
</tr>
<tr>
<td>Alt-D</td>
<td>Debug menu</td>
<td>Takes you to the Debug menu</td>
</tr>
<tr>
<td>Alt-E</td>
<td>Edit menu</td>
<td>Takes you to the Edit menu</td>
</tr>
<tr>
<td>Alt-F</td>
<td>File menu</td>
<td>Takes you to the File menu</td>
</tr>
<tr>
<td>Alt-H</td>
<td>Help menu</td>
<td>Takes you to the Help menu</td>
</tr>
<tr>
<td>Alt-O</td>
<td>Options menu</td>
<td>Takes you to the Options menu</td>
</tr>
<tr>
<td>Alt-P</td>
<td>Project menu</td>
<td>Takes you to the Project menu</td>
</tr>
<tr>
<td>Alt-R</td>
<td>Run menu</td>
<td>Takes you to the Run menu</td>
</tr>
<tr>
<td>Alt-S</td>
<td>Search menu</td>
<td>Takes you to the Search menu</td>
</tr>
<tr>
<td>Alt-W</td>
<td>Window menu</td>
<td>Takes you to the Window menu</td>
</tr>
<tr>
<td>Alt-X</td>
<td>File</td>
<td>Quit</td>
</tr>
</tbody>
</table>
### Editing hot keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-Del</td>
<td>Edit</td>
<td>Clear</td>
</tr>
<tr>
<td>Ctrl-Ins</td>
<td>Edit</td>
<td>Copy</td>
</tr>
<tr>
<td>Shift-Del</td>
<td>Edit</td>
<td>Cut</td>
</tr>
<tr>
<td>Shift-Ins</td>
<td>Edit</td>
<td>Paste</td>
</tr>
<tr>
<td>Ctrl-L</td>
<td>Search</td>
<td>Search Again</td>
</tr>
<tr>
<td>Alt-S R</td>
<td>Search</td>
<td>Replace</td>
</tr>
<tr>
<td>Alt-S F</td>
<td>Search</td>
<td>Find</td>
</tr>
<tr>
<td>F2</td>
<td>File</td>
<td>Save</td>
</tr>
<tr>
<td>F3</td>
<td>File</td>
<td>Open</td>
</tr>
</tbody>
</table>

### Window management hot keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt-#</td>
<td>Window</td>
<td>List</td>
</tr>
<tr>
<td>Alt-0</td>
<td>Window</td>
<td>Close</td>
</tr>
<tr>
<td>Alt-F3</td>
<td>Window</td>
<td>User Screen</td>
</tr>
<tr>
<td>Alt-F4</td>
<td>Debug</td>
<td>Inspect</td>
</tr>
<tr>
<td>Alt-F5</td>
<td>Window</td>
<td>Zoom</td>
</tr>
<tr>
<td>F6</td>
<td>Window</td>
<td>Next</td>
</tr>
<tr>
<td>Ctrl-F5</td>
<td>Window</td>
<td>Next</td>
</tr>
</tbody>
</table>

### Online Help hot keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help</td>
<td>Contents</td>
</tr>
<tr>
<td>F1 F1</td>
<td></td>
<td>Brings up Help on Help. (Just press F1 when you’re already in the help system.)</td>
</tr>
<tr>
<td>Shift-F1</td>
<td>Help</td>
<td>Index</td>
</tr>
<tr>
<td>Alt-F1</td>
<td>Help</td>
<td>Previous Topic</td>
</tr>
<tr>
<td>Ctrl-F1</td>
<td>Help</td>
<td>Topic Search</td>
</tr>
</tbody>
</table>
**Debugging/Running hot keys**

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt-F4</td>
<td>Debug</td>
<td>Opens an Inspector window</td>
</tr>
<tr>
<td>Alt-F7</td>
<td>Search</td>
<td>Takes you to previous error</td>
</tr>
<tr>
<td>Alt-F8</td>
<td>Search</td>
<td>Takes you to next error</td>
</tr>
<tr>
<td>Alt-F9</td>
<td>Compile</td>
<td>Compiles to .OBJ</td>
</tr>
<tr>
<td>Ctrl-F2</td>
<td>Run</td>
<td>Resets running program</td>
</tr>
<tr>
<td>Ctrl-F3</td>
<td>Debug</td>
<td>Brings up call stack</td>
</tr>
<tr>
<td>Ctrl-F4</td>
<td>Debug</td>
<td>Evaluates an expression</td>
</tr>
<tr>
<td>Ctrl-F7</td>
<td>Debug</td>
<td>Adds a watch expression</td>
</tr>
<tr>
<td>Ctrl-F8</td>
<td>Debug</td>
<td>Sets or clears conditional breakpoint</td>
</tr>
<tr>
<td>Ctrl-F9</td>
<td>Run</td>
<td>Runs program</td>
</tr>
<tr>
<td>F4</td>
<td>Run</td>
<td>Runs program to cursor position</td>
</tr>
<tr>
<td>F7</td>
<td>Run</td>
<td>Executes tracing into functions</td>
</tr>
<tr>
<td>F8</td>
<td>Run</td>
<td>Executes skipping function calls</td>
</tr>
<tr>
<td>F9</td>
<td>Compile</td>
<td>Makes (compiles/links) program</td>
</tr>
</tbody>
</table>

**Full Menus On and Off**

There are two sets of menus in Turbo C++: a full set and a smaller set. You can switch between the two sets by choosing the Options | Full Menus command, which toggles between On and Off. Full menus Off provides the minimum command set you'll need for programming in Turbo C++.

When you run TCINST, you can choose which command set you want as the default. If you're ready to roll up your sleeves and take advantage of all the sophisticated features of Turbo C++, you can turn Full menus On.

When you're working with Full menus On, you'll see additional commands in most menus and additional options in many dialog boxes. For example, here's the difference between the Compile menu with Full menus off and on:

**Full menus off:**

- **Compile**
  - Make EXE file
  - Build all

**Full menus on:**

- **Compile**
  - Compile to OBJ
  - Make EXE file
  - Link EXE file
  - Build all
  - Remove messages

*Full menus only*

To turn Full menus on or off, choose Options | Full Menus.

This manual describes all the menus and dialog box options available in the Full menu set. Where there might be some
confusion, we'll display this text in the margin to alert you that a command is available only when Full menus are on.

---

**Turbo C++ windows**

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

You can have any number of windows open in Turbo C++ (memory allowing), 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.)

Turbo C++ makes it easy to spot the active window by placing a double-lined border around it. The active window always has a close box, a zoom box, scroll bars, and a resize corner. 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 resize corner
- 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, a * will appear to the left of the column and line numbers.

This is what a typical window looks like:
The **title bar**, the topmost horizontal bar of a window, contains the name of the window and the window number. You can double-click the title bar to zoom the window. You can also drag the title bar to move the window around.

The **close box** of a window is the box in the upper left corner. You click this box to quickly close the window. (You can also choose **Window | Close** or press **Alt-F3**.) The Inspector and Help windows are considered temporary and can be closed by pressing **Esc**.

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

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.

The resize box is in the lower right corner of a window. You drag any corner to make the window larger or smaller. You can spot the resize corner by its single-line border instead of the double-line border used in the rest of the window. To resize using the keyboard, choose Size/Move from the Window menu, or press Ctrl-F5.

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. If you click the ↓, the window will return to its previous size. To zoom a window from the keyboard, choose Window | Zoom, or press F5.

The first nine windows you open in Turbo 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 (topmost) 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, you can press Alt-5 to quickly bring it to the front.

Window management

Table 1.1 gives you a quick rundown of how you handle windows in Turbo C++. Note that you don’t need a mouse to perform these actions—a keyboard works just fine.

<table>
<thead>
<tr>
<th>To accomplish this:</th>
<th>Use one of these methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open an Edit window</td>
<td>Choose File</td>
</tr>
<tr>
<td>Open other windows</td>
<td>Choose the desired window from the Window menu</td>
</tr>
<tr>
<td>Close a window</td>
<td>Choose Close from the Window menu (or press Alt-F3), or click the close box of the window.</td>
</tr>
<tr>
<td>Activate a window</td>
<td>Click anywhere in the window, or Press Alt plus the window number (1 to 9, in the upper right border of the window), or Choose Window</td>
</tr>
</tbody>
</table>
The status line appears at the bottom of the Turbo C++ screen. The status line functions like this:

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

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

| TIP | TIP | TIP |

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 Options menu title is selected.
If you have a color monitor, Turbo C++ will use different colors for various elements of the dialog box.

This dialog box has three standard buttons: OK, Cancel, and Help. If you choose OK, the choices in the dialog box are made in Turbo C++; 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, you can just click the button you want. When you’re using the keyboard, you can 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 from one item to another in a dialog box. Each element highlights 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.) Be aware that tabbing to a button makes that button the default.
Check boxes and radio buttons

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. You check a check box (set it to on) by clicking 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, Turbo C++ indicates the active check box or group of check boxes by placing a chevron symbol (») next to it. When you press Tab, the chevron moves to the next group of checkboxes or radio buttons.

Radio buttons are so called because they act just like the group of buttons on a real-world 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.

The dialog box also has radio buttons. 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.

Here's what some check boxes and radio buttons look like on and off:

[X] Standard stack frame  ( ) None
[ ] Test stack overflow  (•) Emulation
( ) 8087

Input boxes and lists

Dialog boxes can also contain input boxes. These boxes 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 is useful for search strings.
If an input box has a down-arrow icon to its right, there is a *history list* associated with that input box. You 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 *Down arrow* 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("\n\nprintf(
char buf[7]
*/
return(abortit
return(ABORTIT
```

A final component of many dialog boxes is a *list box*. A list box lets you scroll through and select from variable-length lists 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 Turbo 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 new editing features should help you identify the areas that have changed.

Turbo C++’s integrated editor now has

- mouse support
support for large files (greater than 64K; limited to 8 megabytes for all editors combined)

- Shift ↑ ↓ → ← for selecting text

- Edit windows that you can move, resize, or overlap

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

- multiple windows that let you have several views onto the same file or different files

- a sophisticated macro language, so you can create your own editor commands

- the ability to paste text or examples from the Help window

- an editable Clipboard allowing for 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 Turbo C++

Part 3: Menu reference

This section contains a description of each menu command in Turbo C++. It is arranged by menus. If a command name has “Full menus” next to it, this command appears only when the Options | Full Menus command is on.

≡ (System) menu

Alt Spacebar

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 (About, Clear Desktop, Repaint Desktop) and the names of programs you’ve installed with the Options | Transfer command.

About

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

Clear Desktop

Choose ≡ | Clear Desktop to close all windows and clear all history lists. This command is useful when you’re starting a new project.
Repaint Desktop

Choose **Repaint Desktop** to have Turbo 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 (Options | Debug | Display swapping) and you’re stepping through a program.

Transfer items

Any programs you’ve installed with the Transfer dialog box (Options | Transfer) 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 Options | Transfer.

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 press Enter.

File menu

The File 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

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:

![The Load a File dialog box]

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:
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.

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.

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, Turbo C++ opens it. (If you enter a file name that Turbo 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.

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.

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.
File | New

New

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

Save

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 (NONAME\text{00}.C, or the like), Turbo 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 File | Save As 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:

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 File | Save All command works just like the Save 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.
The **File | Change Dir** command lets you specify a drive and a directory to make current. The current directory is the one Turbo C++ uses to save files and to look for files. (When using relative paths in **Options | Directories**, they are relative to this current directory only.)

Here is what the Change Directory dialog box looks like:

![Change Directory dialog box](image)

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 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 | Print** command lets you print the contents of the active Edit window. Turbo 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 | Get Info command displays a box with information on the current file.

![The Get Info box](image)

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>
<thead>
<tr>
<th>Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current directory</td>
<td>The default directory</td>
</tr>
<tr>
<td>Current file</td>
<td>File in the active window</td>
</tr>
<tr>
<td>Extended memory usage</td>
<td>Amount of extended memory reserved by Turbo C++</td>
</tr>
<tr>
<td>Expanded memory usage</td>
<td>Amount of expanded memory reserved by Turbo C++</td>
</tr>
<tr>
<td>Lines compiled</td>
<td>Number of lines compiled</td>
</tr>
<tr>
<td>Total warnings</td>
<td>Number of warnings issued</td>
</tr>
<tr>
<td>Total errors</td>
<td>Number of errors generated</td>
</tr>
<tr>
<td>Total time</td>
<td>Amount of time your program has run</td>
</tr>
<tr>
<td>Program loaded</td>
<td>Debugging status</td>
</tr>
<tr>
<td>Program exit code</td>
<td>DOS termination code of last terminated program</td>
</tr>
<tr>
<td>Available memory</td>
<td>Amount of free DOS (640K) memory</td>
</tr>
<tr>
<td>Last step time</td>
<td>Amount of time spent in last debug step</td>
</tr>
</tbody>
</table>

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

DOS Shell

The File | DOS Shell command lets you temporarily exit Turbo C++ to enter a DOS command or program. To return to Turbo 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 | Program Reset (Ctrl-F2).
Warning: Don’t install any TSR programs (like SideKick) if you’ve shelled to DOS, because memory may get misallocated.

Note: In dual monitor mode, the DOS command line appears on the Turbo 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, Window | User 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 Turbo C++.

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

Edit menu

The Edit menu lets you cut, copy, and paste text in Edit windows. You can also open a Clipboard window to view or edit its contents.

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:

New!  ■ 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 Edit menu become available, and the Clipboard becomes useful.

The Clipboard is the magic behind cutting and pasting. It’s a special window in Turbo 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 Edit menu.

Here’s an explanation of each command in the Edit menu.

- **Restore Line**

  The Edit | Restore Line command takes back the last editing command you performed on a line. Restore Line works only on the last modified or deleted line.

- **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.

- **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.

- **Copy Example**

  The Edit | Copy Example 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.
The **Edit | Show 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 Turbo C++ uses when you choose Paste.

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. Turbo C++ uses whatever text is selected in the Clipboard when you choose Paste.

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 Cut or Copy, 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.)

The **Edit | 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 Cut or Copy. The cleared text is not retrievable.

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

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 1.8
The Find dialog box

The Find dialog box contains several buttons and check boxes:

- **[ ] Case sensitive**
  Check the Case Sensitive box if you do want Turbo C++ to differentiate uppercase from lowercase.

- **[ ] Whole words only**
  Check the Whole Words Only box if you want Turbo 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 Turbo 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`. 

Turbo C++ User's Guide
[ - ] 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 Turbo 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 Find from the Search menu. You can take additional characters from the text by pressing →.

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

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.

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

The Search | Replace 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.
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 to 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 Turbo 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.

Like in 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 Find or Replace from the Search menu. And you can add more text from the Edit window by pressing →.

The Search | Search Again command repeats the last Find or Replace command. All settings you made in the last dialog box used (Find or Replace) remain in effect when you choose Search Again.

The Search | Go to Line Number command prompts you for the line number you want to find.

Here is what the dialog box looks like:
Turbo C++ displays the current line number and column number in the lower left corner of every Edit window.

**Previous Error**

The Search | Previous 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**

The Search | Next 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 Search | Locate 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.

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

**Run menu**

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

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 integrated environment; see Chapter 2, “Managing multi-file projects,” for more on this feature.)

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 Options | Debugger 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:

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.

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 TC to stop execution on the next source line in your program. If TC is unable to find a source line, a second Ctrl-Break will terminate the program and return you to the IDE.
Program Reset

The **Run | Program 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

The **Run | Go 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 | Go 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 **Go to Cursor** command again.

Use **Go 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

The **Run | Trace 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 | Step Over**). If a statement contains no calls to functions accessible to the debugger, **Trace Into** stops at the next executable statement.

Use the **Trace 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 **Trace Into** and **Step Over** commands.

If the statement contains a call to a function accessible to the debugger, **Trace Into** halts at the beginning of the function’s definition. Subsequent **Trace Into or Step 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,

```c
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 Trace Into command recognizes only functions defined
in a source file compiled with two options set on:

- In the Code Generation dialog box (Options | Compiler), the
  Debug Info in OBJs check box must be checked.
- The Source Debugging radio buttons must be set to On (in the
  Options | Debugger dialog box).

The Run | Trace Into 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 Step 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 Run | Trace Into and
Run | Step Over. These are the first 12 lines of a program loaded
into an Edit window:

```c
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 | Trace 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 | Step 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 | Trace Into and Run | Step 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 | Arguments 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.

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

Compile menu

Use the commands on the Compile menu to compile the program in the active window or to make or build your project. To use the Compile, Make, Build, and Link commands, you must have a file open in an active Edit window or a project defined (for Make, Build, and Link). For example, if you open a Message or Watch window, those selections will be disabled.
Compile | Compile to OBJ

Compile to OBJ  The Compile | Compile 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 Turbo 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  The Compile | Make 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:
- the project file (.PRJ) specified with the Project | Open 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)

Make EXE File rebuilds only the files that aren’t current.

Link EXE File  The Compile | Link 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 Compile | Build All command rebuilds all the files in your project regardless of whether they’re out of date.

This command is similar to Compile | Make EXE File except that it is unconditional. The Build All command first sets the date and time of all the project’s .OBJ files to zero, then does a make. (Thus, if you abort a Build All command by pressing Ctrl-Break or get errors that stop the build, you can pick up where it left off simply by choosing Compile | Make EXE File.)
The **Compile | Remove Messages** command removes all messages from the Message window.

### Debug menu

The commands on the **Debug** menu control all the features of the integrated debugger. You can change default settings for these commands in the **Options | Debugger** dialog box.

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 Turbo 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 are displayed. For example,
you can range-inspect pointer variables to tell Turbo 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-l.

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

```c
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

```c
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

```c
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 and union data items. For example,

```c
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 as 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.

Debug | Evaluate/Modify

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:

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 \texttt{xarray},

- \texttt{xarray[0],5} displays five consecutive integers in decimal.
- \texttt{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 Debug \textbf{I} Evaluate/Modify command displays each type of value in an appropriate format. For example, it displays an \texttt{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 1.3.

The Debug \textbf{I} Call 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 \texttt{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.

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 \texttt{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:

\begin{verbatim}
func2()
func1()
main()
\end{verbatim}

This tells you that \texttt{main} called \texttt{func1}, and \texttt{func1} called \texttt{func2}. If you wanted to see the currently executing line of \texttt{func1}, you could select \texttt{func1} in the call stack and press \texttt{Enter}. The code for \texttt{func1} would appear in the Edit window, with the cursor positioned on the call to \texttt{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 \texttt{Enter}.
Table 1.3: Format specifiers recognized in debugger expressions

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td><strong>Character.</strong> 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.</td>
</tr>
<tr>
<td>S</td>
<td><strong>String.</strong> 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 <strong>S</strong> specifier is only useful in conjunction with the <strong>M</strong> specifier.</td>
</tr>
<tr>
<td>D</td>
<td><strong>Decimal.</strong> Shows all integer values in decimal. Affects simple integer expressions as well as arrays and structures containing integers.</td>
</tr>
<tr>
<td>H or X</td>
<td><strong>Hexadecimal.</strong> Shows all integer values in hexadecimal with the 0x prefix. Affects simple integer expressions as well as arrays and structures containing integers.</td>
</tr>
<tr>
<td>Fn</td>
<td><strong>Floating point.</strong> Shows ( n ) significant digits (( n ) is an integer between 2 and 18). The default value is 7. Affects only floating-point values.</td>
</tr>
<tr>
<td>M</td>
<td><strong>Memory dump.</strong> 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 <strong>M</strong> specifier is ignored. By default, each byte of the variable is shown as two hex digits. Adding a <strong>D</strong> specifier with the <strong>M</strong> causes the bytes to be displayed in decimal. Adding an <strong>H</strong> or <strong>X</strong> specifier causes the bytes to be displayed in hex. An <strong>S</strong> or a <strong>C</strong> 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.</td>
</tr>
</tbody>
</table>
| P         | **Pointer.** Displays pointers in \( \text{seg:ofs} \) format with additional information about the address pointed to, rather than the default hardware-oriented \( \text{seg:ofs} \) 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:

<table>
<thead>
<tr>
<th>Memory region</th>
<th>Evaluate message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0000-0000:03FF</td>
<td>Interrupt vector table</td>
</tr>
<tr>
<td>0000:0400-0000:04FF</td>
<td>BIOS data area</td>
</tr>
<tr>
<td>0000:0500-Turbo C++</td>
<td>MS-DOS/TSR's</td>
</tr>
<tr>
<td>Turbo C++—User Program PSP</td>
<td>Turbo C++</td>
</tr>
<tr>
<td>User Program PSP</td>
<td>User Process PSP</td>
</tr>
<tr>
<td>User Program—top of RAM</td>
<td>Name of a static user variable if its address falls inside the variable's allocated memory; otherwise nothing</td>
</tr>
<tr>
<td>A000:0000-AFFF:FFFF</td>
<td>EGA/VGA Video RAM</td>
</tr>
<tr>
<td>B000:0000-B7FF:FFFF</td>
<td>Monochrome Display RAM</td>
</tr>
<tr>
<td>B800:0000-BFFF:FFFF</td>
<td>Color Display RAM</td>
</tr>
<tr>
<td>C000:0000-EFFF:FFFF</td>
<td>EMS Pages/Adaptor BIOS ROM's</td>
</tr>
<tr>
<td>F000:0000-FFFF:FFFF</td>
<td>BIOS ROM's</td>
</tr>
<tr>
<td>R</td>
<td><strong>Structure/Union.</strong> Displays field names as well as values, such as { X:1, Y:10, Z:5 }. Affects only structures and unions.</td>
</tr>
</tbody>
</table>
Debug | Watches

The **Debug | Watches** command opens a pop-up menu of commands that control the use of watchpoints. The following sections describe the commands in this pop-up menu.

### Add Watch

The **Add 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 **Delete Watch** command deletes the current watch expression from the Watch window.

The Watch window must be the active window in order to use this command. The current watch expression is selected if the Watch window is active. It's marked by a bullet in the left margin if the Watch window is inactive and instead another window, menu, or dialog box is active.

To delete the watch expression marked with the bullet, choose the **Delete Watch** command. To delete a watch expression that is not current (that is, one that is not selected or has no bullet), you must make the Watch window active, select the desired watch expression, and press either *Del* or *Ctrl-Y*.

### Edit Watch

The **Edit 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 Remove All Watches command deletes all watch expressions from the Watch window.

The Debug | Toggle 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. See the following section for more information on breakpoints.

The Debug | Breakpoints 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:

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:
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 are lost only when

- you leave the integrated environment
- you delete the source line a breakpoint is set on
- you clear a breakpoint with Toggle Breakpoint

Turbo 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. (Turbo 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, Turbo 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, Turbo 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 **Debug | Toggle Breakpoint** command.

---

### Project menu

The **Project menu** contains all the project management commands to

- create a project
- add or delete files from your project
- 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 **Open 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.

This dialog box lets you select a file name similar to the **File | Open** dialog box, discussed on page 22. 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. Turbo 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 I Close Project** when you want to remove your project and return to the default project (TCDEF.DPR).

Add Item
Choose **Project I Add Item** when you want to add a file to the project list. This brings up the Add Item to Project List dialog box, which looks like this:

![Add Item to Project List dialog box](image)

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 I 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 a dialog box, which looks like this:
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 64 for information on the Transfer dialog box).

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 Options | Compile | Code Generation).

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).

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

Include Files Choose Project | Include 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 dialog box](image)

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 Project menu and choose Include 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

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

**Full Menus**

The **Options | Full Menus** command lets you use a subset of the complete set of menus in Turbo C++. **Full Menus Off** provides the minimum command set in menus and dialog boxes. (Use `INSTALL` to set up which command set you want to use as the default.) For more information on this command, see page 13.

**Compiler**

The **Options | Compiler** 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 Code 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 (if you have Full Menus set to Off, some of the options in this box won’t appear):

Here are what the various buttons and check boxes mean:

- Checking Overlay Support tells the compiler to generate overlay safe code. You should check this (turn this on) when you’re running an overlaid application. This is a global option; it controls whether Overlay this Module (in the Override Options dialog box of Project | Local Options) and Overlay EXE (Options | Linker) are enabled or disabled.

- Word Alignment (when checked) tells Turbo C++ to align noncharacter data (structs and unions only) at even addresses. When this option is off (unchecked), Turbo C++ uses byte-aligning, where data (structs 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 8086 and 80286 processors fetch and store the data.

- Duplicate Strings Merged (when checked) tells Turbo 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 Turbo C++ to treat all char declarations as if they were unsigned char type. It’s checked by default.

- 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. The default is off (unchecked).
If a source file is compiled 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 Debug | Call Stack command from "seeing" the function. Thus, the option should always be checked when a source file is compiled for debugging.

- Test Stack Overflow (when checked) 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. The default is off (unchecked).

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 4, "Memory models, floating point, and overlays," in the Programmer's Guide for more information about memory models.

Use the Defines input box to enter macro definitions to the preprocessor. You can separate multiple defines with semicolons (;); for example, TESTCODE; PROGCONST=5

Values can be assigned optionally with an equal sign (=).

Leading and trailing spaces are 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.

If you have Full Menus on when you select the Code Generation command, 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:
The Floating Point buttons let you decide how you want Turbo 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 Turbo C++ to detect whether your computer has an 80x87 coprocessor (and to use it if you do). If it is not present, Turbo C++ emulates the 80x87.

Choose 8087 to generate direct 8087 inline code.

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

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

Do not change this option unless you’re an expert and have read Chapter 6, “Interfacing with assembly language,” in the Programmer’s Guide.
When checked, the Generate Underbars option tells Turbo C++ to automatically add 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 map file (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. The default is off (unchecked).

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 should make sure the Jump Optimization check box 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 is needed for you to do both integrated debugging and debugging with the standalone Turbo Debugger.

- When checked, Treat enums as ints causes the compiler to always allocate a whole word. This option is checked by default. (The command-line equivalent is `-b`.)

- Fast Floating Point lets you optimize floating-point operations without regard to explicit or implicit type conversions. This option is checked by default. (You can use `-ff` to accomplish the same thing on the command line.)

- When checked, Assume SS Not Equal DS option causes the compiler to not assume the stack segment (SS) to be equal to the data segment (DS). This option is unchecked by default. (The command-line equivalent is `-mm!`.)

**C++ Code Generation**

The C++ Code Generation command displays a dialog box that contains settings that tell the compiler to prepare the object code in certain ways when using C++.
The C++ Virtual Tables radio buttons let you control C++ virtual tables and the expansion of inline functions when debugging.

Choosing 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 command-line equivalent is -V.)

Choosing 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 command-line equivalent is -Vs.)

Choosing 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 command-line equivalent is -VO.)

Choosing the Public option generates public definitions for virtual tables, so that these can be externally referenced in other modules that have been compiled with the External option. (The command-line equivalent is -V1.)

The Use C++ Compiler radio buttons tell Turbo 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.

You’ll use the Out-of-Line Inline Functions when you want to step through or set breakpoints on inline functions.

The Optimizations 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 (available only on Full menus):
The check boxes in 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 because the compiler cannot 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.

  When this option is checked, the sequences of tracing and stepping in the integrated 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.

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 4, “Memory models, floating point, and overlays,” 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.
The Optimize For buttons let you change Turbo 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.

Source

The Source 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:

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

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

- Choosing Turbo C++ tells the compiler to recognize the Turbo C++ extension keywords, including near, far, huge, asm, cdecl, pascal, interrupt, _es, _ds, _cs, _ss, and the register pseudovariables (_AX, _BX, and so on). For a complete list, refer to to Chapter 1, “The Turbo C++ language standard,” in the Programmer's Guide.

- Choosing ANSI tells the compiler to recognize only ANSI keywords and treat any Turbo C++ extension keywords as normal identifiers.

- Choosing UNIX V tells the compiler to recognize only UNIX V keywords and treat any Turbo C++ extension keywords as normal identifiers.

- Choosing Kernighan and Ritchie tells the compiler to recognize only the K&R extension keywords and treat any Turbo C++ extension keywords as normal identifiers.
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 Messages command displays a dialog box that lets you set several options that affect compiler error messages in the integrated environment.

If Full Menus is on, the dialog box also has four buttons that lead to six separate dialog boxes. Each of the nested dialog boxes lets you turn on or off individual types of error messages.

Here is what the full dialog box looks like:

- 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.

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. Here is what this dialog box looks like:

![Portability Warning](image)

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 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:

![ANSI Violations](image)

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. Here is what this dialog box looks like:
Check or uncheck these warnings just like in the previous dialog box's and choose OK to return to the ANSI Violations dialog box.

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:

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 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:
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, which looks like this:

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

**Names**

The Names command brings up a dialog box that 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 4, “Memory models, floating point, and overlays,” in the Programmer’s Guide.*

The dialog box looks like this:
The Transfer command lets you add or delete programs in the \texttt{=} menu. Then you can choose items from this menu to run another program without actually leaving Turbo C++. You return to Turbo C++ only after you exit the program you transferred to.

The Transfer command displays a dialog box that looks like this:

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

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 Save 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.)

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 Turbo C++.

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
Options | Transfer

*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.

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

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

Transfer macros

The IDE recognizes certain macro names in the parameter string of the Modify/New Transfer Item dialog box. There are three kinds of macros:

- state
- file name
- instruction

Macros are expanded based on the state of the IDE. For example, the macro $EDNAME refers to the file in the currently active editor window; $EXENAME refers to the program that has been or will be generated, as shown on the *Compile* menu.

The file name macros are actually functions that let you access parts of file name specifications. For example, you may want a macro that calls TDUMP and always dumps the object file of the corresponding file in the editor, which requires stripping off the path and extension and adding on the output path and the .OBJ extension.

Instruction macros tell the integrated environment to perform some action or make some setting.

Here's a look at what you can do with the macros available.
TEML is a Pascal-like language that has many built-in primitive editor commands. See Appendix A, "Turbo Editor macros" for more information on it.

State macros

$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.

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.

$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.

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

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

File name macros

These macros take file names as arguments and return various parts of the file name. This allows you to build up new file name specifications from existing file names. For example, you can pass TDUMP a macro like this:

\[ S\text{DIR}($\text{EXENAME}$) S\text{NAME}($\text{EDNAME}$) .OBJ \]

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

$\text{DIR}()$: Directory of the file argument, full path with trailing backslash; for example, \turboc\.

$\text{DRIVE}()$: Drive of the file argument, in the form D:.

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

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

Instruction macros

$\text{CAP EDIT}$: 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 $\text{CAP EDIT}$ to work correctly, the transfer program must write to DOS standard output.

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

Two filters are provided: TASM2MSG.EXE for Turbo Assembler (TASM) and GREP2MSG.EXE for GREP. The source code to these filters is also provided so you can write your own filters for other transfer programs you install.

Any program that has line-oriented messages output (file and line number) could be used with this macro.
$MEM(): 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.

$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.

$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.

$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 $EDNAME,$DRIVE($EXENAME)$DIR($EXENAME)$NAME($EDNAME).OBJ
$CAP MSG(TASM2MSG)
```

Transfer memory setting

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 $MEMO 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 Program Reset (Ctrl-F2) to free up debugging memory.

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

The Options | Make command displays a dialog box that lets you set conditions for project management. Here's what the dialog box looks like:

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

When the Check Autodependencies 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 Turbo C++ and the command-line version of Turbo 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 .C 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.

The **Options | Linker** command lets you make several settings that affect linking. The Linker command opens a dialog box, which looks like this:

![The Linker dialog box](image)

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

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

When you’re linking with modules created by a compiler other than Turbo 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 Turbo C++. If this option is off (unchecked), the linker searches only the default libraries supplied by Turbo C++ and ignores any defaults in .OBJ files.

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.
Options | Linker

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.

The Warn Duplicate Symbols option affects whether the linker warns you of previously encountered symbols in .LIB files. The default is off (unchecked).

The Stack Warning option affects whether the linker generates the "No stack" message.

Note: It's normal for a program generated under the tiny model to display this message if the message is not turned off.

The Case-Sensitive Link option affects whether the linker is case-sensitive. Normally, this option should be checked, since C is a case-sensitive language.

When you check the Overlay EXE option, this tells the linker to overlay the program. This lets you relink to switch between overlaid and not overlaid. This is a global option. When it's unchecked (turned off), Turbo C++ produces no overlays regardless of the overlay setting in the Override Options dialog box (Project | Local Options).

The Options | Debugger command lets you make several settings affecting the integrated debugger. This command opens a dialog box, which looks like this:

The following sections describe the contents of this box.

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

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.

**Note:** 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 Turbo C++ command-line /d option), you can see your program's output on one monitor and the Turbo C++ screen on the other. In this case, Turbo C++ never swaps screens and the Display Swapping setting has no effect.
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.

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

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.

The Options | Directories command lets you tell Turbo C++ where to find the files it needs to compile, link, and output binary and map files.

This command opens a dialog box containing three input boxes. The dialog box looks like this:

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 Turbo C++ startup object files (C0?.OBJ) and runtime 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. Turbo 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:TC\MATHLIBS;A:..\VIDLIBS

Environment

The Options | Environment command lets you make environment-wide settings. This command opens a menu that lets you choose settings from Preferences, Editor, and Mouse. (If you have Full Menus off, you won't see all these options.)

Here's what the Preferences dialog box looks like:

The Screen Size radio buttons let you specify whether your integrated environment 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), Turbo 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.

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.

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

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

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

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

When Save Old Messages is checked, Turbo 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. When you uncheck this box, Turbo C++ automatically clears messages before a compile, a make, or a transfer that uses the Message window.

If you choose Editor from the Environment menu, these are the options you can pick from:
The Editor Options dialog box has several check boxes that control how Turbo C++ handles text in Edit windows.

- When Create Backup Files is checked (the default), Turbo C++ automatically creates a backup of the source file in the Edit window when you choose File -> Save 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 non-blank 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, Turbo C++ inserts a true tab character (ASCII 9) when you press Tab. When this option is not checked, Turbo C++ replaces tabs with spaces, the number of which is determined by the Tab Size setting, discussed later.

- When you check Optimal Fill, Turbo 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.

If you check Use Tab Character in this dialog box and press Tab, Turbo 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. Turbo 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 Options -> Save Options.
The Default Extension input box lets you tell Turbo C++ which extension to use as the default when compiling and loading your source code.

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

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 Help | Topic Search
- Go to Cursor: Same as Run | Go To Cursor
- Breakpoint: Same as Debug | Toggle Breakpoint
- Inspect: Same as Debug | Inspect
- Evaluate: Same as Debug | Evaluate
- Add Watch: Same as Debug | Watches | Add Watch

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 Turbo C++ requires a shorter time between clicks to recognize a double click. Moving the scroll box closer to Slow means Turbo 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. The bar highlights when you successfully double-click it.

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.
The Options | Save command brings up a dialog box that lets you save settings that you've made in both the Find and Replace dialog boxes (off the Search menu) and in the Options menu (which includes all the dialog boxes that are part of those commands) for Environment, Desktop, and Project items. Options are stored in three files, which represent each of these categories.

If it doesn't find the files, Turbo C++ looks in the Executable directory (where TC.EXE is run from) for the same file.

### Window menu

The Window 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 14 for information on these elements and how to use them.

At the bottom of the Window menu, the Window | List 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.)

Choose Window | Size/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.

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

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

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

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

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 **Window | Close**.

Choose **Window | Output** 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 **Options | Environment** 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 **Window | User Screen** command instead.

To close the window, click its close box or choose **Window | Close**.
Watch

Choose Window | Watch 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 Debug | Watches 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 45).

To close the window, click its close box or choose Window | Close.

User Screen

Choose Window | User Screen to view your program’s full-screen output. If you would rather see your program output in a Turbo C++ window, choose the Window | Output command instead.

Clicking or pressing any key returns you to the integrated environment.

Register

Choose Window | Register 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 Window | Close.

Project

Choose Window | Project to open the Project window, which lets you view files that you’re using to create your program.

Notes

Choose Window | Notes to write down any details, make to-do lists, or list any other information about your project files.

List

Choose Window | List 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, Turbo C++ brings the window to the front and makes it active. When you choose a closed file from the list, Turbo 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.
Press \textit{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 16.

---

### Help menu

The Help menu gives you access to online help in a special window. There is help information on virtually all aspects of the integrated environment and Turbo 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:

- Press \textit{F1} at any time (including from any dialog box or when any menu command is selected).
- When an Edit window is active and you’re positioned on a word, press \textit{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 \textit{Esc}, click the close box, or choose \texttt{Window} \textbar \texttt{Close}. 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 \textit{F1} when a menu command was selected. (If you press \textit{F6} or click on another window while you’re in Help, the Help window remains onscreen.)

Help screens often contain \textit{keywords} (highlighted text) that you can choose to get more information. Press \textit{Tab} to move to any keyword; press \textit{Enter} to get more detailed help. (As an alternative, move the cursor to the highlighted keyword and press \textit{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 \textit{Ctrl-F1} on \textit{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 \textit{Shift→}, Left arrow, Up arrow, Down arrow), choose \texttt{Edit} \textbar \texttt{Copy}, move to an Edit window, then choose \texttt{Edit} \textbar \texttt{Paste}. 

---

When getting help in a dialog box or menu, you cannot resize the window or copy to the clipboard. In this instance, \textit{Tab} takes you to dialog box controls, not the next keyword.
To select text in the Help window, drag across the desired text or, when positioned at the start of the block, press \texttt{Shift-\rightarrow}, \texttt{\leftarrow}, \texttt{\uparrow}, \texttt{\downarrow} to mark a block.

You can also copy preselected program examples from help screens by choosing the \texttt{Edit \ Copy Example} command.

The \texttt{Help \ Contents} 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.

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

The \texttt{Help \ Index} 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 \texttt{p r i}. When you type \texttt{p}, the cursor jumps to the first keyword that starts with \texttt{p}. When you then type \texttt{r}, the cursor then moves to the first keyword that starts with \texttt{pr}. When you then type \texttt{i}, the cursor moves to the first keyword that starts with \texttt{pri}, and so on.

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

The \texttt{Help \ Topic 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 \texttt{Topic Search}. You can get help on things like function names (\texttt{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.
Help I Previous Topic

Previous Topic
The Help | Previous Topic command opens the Help window and redisplays the text you last viewed.

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

Project and configuration files
The integrated environment for Turbo C++ 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.

Old-style 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.

The new project file
In Turbo C++, 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 auto-dependency information.
The new configuration file

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

In Turbo C++'s IDE, the TCCONFIG.TC file contains only environmental information. The .TC file is no longer required to build programs defined by a project. The information stored in the .TC file includes

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

Loading project files

Project files are loaded in three ways:

1. When invoking Turbo C++, give the project name with the .PRJ extension after the TC command; for example,

   TC 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 TC 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 Project | Open 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 Options | Directories 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

- the context information for each file in the project (that is, the position in the file, the location of the window on the screen, etc.)
- the history lists for various input boxes (for example, search strings, file masks, etc.)
- layout of the windows on the desktop

Changing project files

Because each project file has its own desktop file, changing to another project file causes the current desktop to be written out and 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 Project | Open Project and typing in a new .PRJ file), the new project’s desktop inherits the previous desktop. When you select Project | Close Project, the default project is loaded and you get the default desktop.

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 TC.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).
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. Turbo 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 Project | Open Project)
- add source files using the Project | Add Item dialog box
- tell Turbo C++ to Compile | Make EXE File

Then, with the project-management commands available on the Project 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

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:

```c
#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:

```c
char ss[] = "The restaurant at the end of";
char *getString(void)
{
  return ss;
}
```

And MYFUNCS.H looks like this:

```c
extern char *getString(void);
```

These files make up the program that we'll now describe to the Project Manager.
The first step is to tell Turbo 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 Project menu and choose Open 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 Add Item, Delete Item, Local Options, and Include Files options are enabled on the Project menu.

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, and F10 takes you to the main menu. Press Ins now to add a file to the project list.

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.

Note that the Add button commits your change: pressing Esc when you're in the dialog box just puts the dialog box away.

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. Turbo C++ will compile files in the exact order they appear in the project.

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.

After all compiler options and directories have been set, Turbo 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.

Press F10 to go to the main menu. Now make MYPROG.EXE by pressing F9 (or choose Compile I Make EXE File). Then run your program by pressing Ctrl-F9 (or choose Run I Run). To view your output, choose Window I User Screen (or press Alt-F5), then press any key to return to the integrated environment.
For more information on .PRJ and .DSK files, refer to the section, “Project and configuration files,” in Chapter 1.

You can specify a project to load on the DOS command line like this: TC myprog.prj.

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 (Options | Environment).

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.

The next time you use Turbo C++, you can jump right into your project by reloading the project file. Turbo 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 Turbo 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.

Since you want to see the effect of tracking in multiple files, you need to modify the criterion Turbo C++ uses to decide when to stop the make process. This is done by setting a radio button in the Make dialog box (Options | 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 (Options | Make). 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.

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), Turbo C++ loads the file it references into
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 (Options | Environment). This way the only messages removed are the ones that result from the files you recompile. 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 Compile | Remove 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 check 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 Turbo C++ integrated environment compiler (TC.EXE) and the Turbo C++ command-line compiler (TCC.EXE) 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 Turbo 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 Turbo 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 (Turbo 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 Turbo C++ integrated environment compiler (TC.EXE) 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:
Except for Turbo C++, each of the names in the Project File Translators list box is a reference to a program defined in the Transfer dialog box (Options | Transfer).

Press Esc, then F10 to return to the main menu, then choose Options | Transfer. 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 Turbo C++ command-line compiler (TCC.EXE) instead of TC.EXE, the integrated environment compiler. To do so, you would perform the following steps:

1. First, you need to define TCC 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 Turbo C++ Command-Line Compiler; in the Program Path input box, type TCC; 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 Turbo 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 Turbo 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 Turbo 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 TCC when compiling MYFUNCS.

MYFUNCS.C now compiles using TCC.EXE, while all of your other source modules compile with TC.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 TC.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 C0x.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 Options | Compiler 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:

```c
#include <stdio.h>
#include "myfuncs.h"
```
```c
#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

```c
extern char *GetString(void);
```

mytime.h contains

```c
#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:

```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
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 Project I Include Items or Alt-P l.) 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 Window I Project 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.
You should read this chapter even if you are familiar with the editor in other Turbo products. Turbo C++'s new IDE includes improvements to the editor. Context-sensitive help is always just a keystroke away (F1).

This chapter is a reference to Turbo C++'s full range of editing commands. Table 3.1 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 a tutorial about the editor and the IDE, refer to Chapter 3 in Getting Started; for an in-depth discussion of the whole Turbo C++ integrated environment, refer to Chapter 1.

The new and the old

The Turbo C++ integrated environment still lets you use Borland's familiar hot key combinations to move around your file, insert, copy, and delete text, and search and replace. However, it also provides you with two brand-new menus on the menu bar, the Edit menu and the Search menu. In addition, Turbo C++ comes with mouse support for 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 Turbo C++ now, an Edit window is already active. To open other Edit windows, go to the File menu and choose Open. 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.

The editor has a restore facility that lets you take back changes to the last line modified. This command (Edit | Restore Line) is described on page 106 in the section titled “Miscellaneous editing commands.”

Editor reference

Table 3.1 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 3.1.

<table>
<thead>
<tr>
<th>Movement</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic cursor movement</td>
<td></td>
</tr>
<tr>
<td>Character left</td>
<td>←</td>
</tr>
<tr>
<td>Character right</td>
<td>→</td>
</tr>
<tr>
<td>Word left</td>
<td>Ctrl←</td>
</tr>
<tr>
<td>Word right</td>
<td>Ctrl→</td>
</tr>
<tr>
<td>Line up</td>
<td>↑</td>
</tr>
<tr>
<td>Line down</td>
<td>↓</td>
</tr>
<tr>
<td>Scroll up one line</td>
<td>Ctrl-W</td>
</tr>
<tr>
<td>Scroll down one line</td>
<td>Ctrl-Z</td>
</tr>
<tr>
<td>Page up</td>
<td>PgUp</td>
</tr>
<tr>
<td>Page down</td>
<td>PgDn</td>
</tr>
</tbody>
</table>

A word is defined as a sequence of characters separated by one of the following: space <> , ; . ( ) { } ^ " " + - / $ # _ = ! ~ ? ! " % & ¡ ; @ { ( ) \ , and all control and graphic characters.
Many of the commands in this table can also be performed with the mouse. See Chapter 1.

<table>
<thead>
<tr>
<th>Movement</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long distance</td>
<td>Movement</td>
</tr>
<tr>
<td>Beginning of line</td>
<td>Home</td>
</tr>
<tr>
<td>End of line</td>
<td>End</td>
</tr>
<tr>
<td>Top of window</td>
<td>Ctrl Home</td>
</tr>
<tr>
<td>Bottom of window</td>
<td>Ctrl End</td>
</tr>
<tr>
<td>Beginning of file</td>
<td>Ctrl PgUp</td>
</tr>
<tr>
<td>End of file</td>
<td>Ctrl PgDn</td>
</tr>
<tr>
<td>Beginning of block</td>
<td>Ctrl-Q B</td>
</tr>
<tr>
<td>End of block</td>
<td>Ctrl-Q K</td>
</tr>
<tr>
<td>Last cursor position</td>
<td>Ctrl-Q P</td>
</tr>
</tbody>
</table>

**Insert and delete commands**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert mode on/off</td>
</tr>
<tr>
<td>Inset</td>
</tr>
<tr>
<td>Delete character left of cursor</td>
</tr>
<tr>
<td>Delete character at cursor</td>
</tr>
<tr>
<td>Delete word right</td>
</tr>
<tr>
<td>Insert line</td>
</tr>
<tr>
<td>Delete line</td>
</tr>
<tr>
<td>Delete to end of line</td>
</tr>
</tbody>
</table>

**Block commands**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark block</td>
</tr>
<tr>
<td>Mark single word</td>
</tr>
<tr>
<td>Copy block</td>
</tr>
<tr>
<td>Move block</td>
</tr>
<tr>
<td>Delete block</td>
</tr>
<tr>
<td>Read block from disk</td>
</tr>
<tr>
<td>Write block to disk</td>
</tr>
<tr>
<td>Hide/display block</td>
</tr>
<tr>
<td>Print block</td>
</tr>
<tr>
<td>Indent block</td>
</tr>
<tr>
<td>Unindent block</td>
</tr>
<tr>
<td>Ctrl-K I</td>
</tr>
<tr>
<td>Ctrl-K U</td>
</tr>
</tbody>
</table>

**Other editing commands**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoindent on/off</td>
</tr>
<tr>
<td>Control character prefix**</td>
</tr>
<tr>
<td>Find place marker</td>
</tr>
<tr>
<td>Go to menu bar</td>
</tr>
<tr>
<td>New file</td>
</tr>
<tr>
<td>Open file</td>
</tr>
<tr>
<td>Optimal fill mode on/off</td>
</tr>
<tr>
<td>Pair matching</td>
</tr>
<tr>
<td>Print file</td>
</tr>
<tr>
<td>Quit IDE</td>
</tr>
<tr>
<td>Repeat last search</td>
</tr>
<tr>
<td>Restore error message</td>
</tr>
</tbody>
</table>
Table 3.1: Full summary of editor commands (continued)

<table>
<thead>
<tr>
<th>Movement</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restore line</td>
<td>Edit</td>
</tr>
<tr>
<td>Return to editor from menus</td>
<td>Esc</td>
</tr>
<tr>
<td>Save</td>
<td>File</td>
</tr>
<tr>
<td>Search</td>
<td>Search</td>
</tr>
<tr>
<td>Search and replace</td>
<td>Search</td>
</tr>
<tr>
<td>Set place marker</td>
<td>Ctrl-K</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Tab mode</td>
<td>Options</td>
</tr>
<tr>
<td>Unindent mode</td>
<td>Options</td>
</tr>
</tbody>
</table>

*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 3.2). 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 3.2: Block commands in depth

<table>
<thead>
<tr>
<th>Movement</th>
<th>Command(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark block</td>
<td>Shift ↓, ↑, →, ←</td>
<td>Marks (highlights) a block as the cursor is moved. Marked text is displayed in a different intensity.</td>
</tr>
<tr>
<td>Mark single</td>
<td>Ctrl-K T</td>
<td>Marks a single word as a block, replacing the block-begin/block-end sequence, which is a bit clumsy for marking a single word. 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.</td>
</tr>
<tr>
<td>Copy block</td>
<td><strong>Edit</strong></td>
<td>Copy, <strong>Ctrl-Ins</strong></td>
</tr>
<tr>
<td>Move block</td>
<td><strong>Edit</strong></td>
<td>Cut, <strong>Shift-Del</strong></td>
</tr>
<tr>
<td>Delete block</td>
<td><strong>Edit</strong></td>
<td>Clear, <strong>Ctrl-Del</strong></td>
</tr>
<tr>
<td>Write block to disk</td>
<td><strong>Ctrl-K W</strong></td>
<td>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. <strong>Note:</strong> 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.</td>
</tr>
<tr>
<td>Read block from disk</td>
<td><strong>Ctrl-K R</strong></td>
<td>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.</td>
</tr>
<tr>
<td>Hide/display block</td>
<td><strong>Ctrl-K H</strong></td>
<td>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.</td>
</tr>
<tr>
<td>Print block</td>
<td><strong>Ctrl-K P</strong></td>
<td>Sends the marked block in the active Edit window to the printer.</td>
</tr>
</tbody>
</table>
Table 3.2: Block commands in depth (continued)

<table>
<thead>
<tr>
<th>Command(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print</td>
<td>Sends the entire file in the active Edit window to the printer.</td>
</tr>
</tbody>
</table>

**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 3.3: Other editor commands in depth

<table>
<thead>
<tr>
<th>Movement</th>
<th>Command(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoindent</td>
<td>Options <img src="text" alt="Environment Editor/" /></td>
<td>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 <em>Enter</em>, 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 <em>Spacebar</em> and ← key to select the new column. Autoindent is on by default.</td>
</tr>
<tr>
<td>Find place marker</td>
<td>Ctrl-Q <img src="text" alt="n" /></td>
<td>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.</td>
</tr>
<tr>
<td>New file</td>
<td>File <img src="text" alt="New" /></td>
<td>Opens a new window.</td>
</tr>
<tr>
<td>Open file</td>
<td>File <img src="text" alt="Open F3" /></td>
<td>Lets you load an existing file into an Edit window.</td>
</tr>
<tr>
<td>Quit edit</td>
<td>File <img src="text" alt="Quit Alt-X" /></td>
<td>Quits Turbo C++. You are asked whether you want to save the file to disk.</td>
</tr>
<tr>
<td>Restore line</td>
<td>Edit <img src="text" alt="Restore Line" /></td>
<td>Lets you undo changes made to the last line worked on. The line is restored to its original state regardless of any changes you have made. This works only on the last modified or deleted line.</td>
</tr>
<tr>
<td>Save file</td>
<td>File <img src="text" alt="Save F2" /></td>
<td>Saves the file and returns to the editor.</td>
</tr>
<tr>
<td>Set place</td>
<td>Ctrl-K <img src="text" alt="n" /></td>
<td>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 <img src="text" alt="n" /> command (being sure to use the same marker number). You can have ten places marked in each window.</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab</td>
<td>Tabs default to eight columns apart in the Turbo C++ editor.</td>
</tr>
<tr>
<td>Tab mode</td>
<td>Options <img src="text" alt="Environment Editor" /></td>
<td>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 8); when it's off, the tab is automatically inserted as the correct number of spaces.</td>
</tr>
</tbody>
</table>

*Turbo C++ User's Guide*
The **Search** \>Find and **Search** \>Replace commands let you search for (and optionally replace) strings of up to 30 characters.

The search string can contain any characters, including control characters. You can enter control characters with the `^P` prefix. For example, enter a `^T` by holding down the `^` key as you press `P` and then `T`. You can include a line break in a search string by specifying `^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.

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. Turbo C++ performs the operation.

5. If you want to search for the same item repeatedly, use **Search** \>Search Again.
Search and replace

1. Choose Search | Replace. 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. Turbo 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 Turbo 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 Turbo 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.

Two match pair commands are necessary because some delimiters are nondirectional.

For example, suppose you tell the editor to find the match for an opening brace ( { ) or an opening square 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.

Table 3.4 Delimiter pairs

<table>
<thead>
<tr>
<th>Delimiter pair</th>
<th>Direction implied?</th>
<th>Are they nestable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( )</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>[ ]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>/* */</td>
<td>Yes</td>
<td>Yes and No</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>' '</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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:

![Diagram of matched pairs](image)

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 3.4, 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 Turbo 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.

- 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 Options | Compiler | Source. 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 3.2, Nested Comments is checked. In Figure 3.3, Nested Comments is unchecked. In the third example, a backward search is performed using Ctrl-Q with Nested Comments still unchecked.

**Figure 3.2**
Forward search I

```
/* /* /* /* Here are some nested comments. */ */ */ */ */
```

```
match level
selected
```

```
match level
found
```

**Note**
A backward search from the found */ will yield the selected */ when Nested Comments is checked.

**Figure 3.3**
Forward search II

```
/* /* /* /* Here are some nested comments. */ */ */ */ */
```

```
match level
selected
```

```
match level
found
```
Figure 3.4
Backward search

/* /* /* /* Here are some nested comments. */ /* /* /* */

[---------------- match level match level ----------------]
selected  found
The Turbo C++ command-line compiler (TCC.EXE) lets you invoke all the functions of the Turbo C++ compiler from the DOS command line.

In addition to using the integrated development environment, you can compile and run your Turbo C++ programs with the command-line interface. While the integrated environment usually is best for developing and running your programs, you may sometimes prefer to use the command line; in some advanced programs, the command-line interface may be the only way to do something intricate. (For instance, MAKE can batch files, while the Project Manager can't.)

TCC compiles C and C++ source files and links them together into an executable file. It works similarly to the UNIX CC command. TCC will also invoke TASM to assemble .ASM source files. Note that to compile only you have to use the -c option at the command line.

To invoke Turbo C++ from the command line, type TCC 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

```shell
tcc [option [option...]] filename [filename...]
```

With two exceptions, each command-line option is preceded by a hyphen (−) and separated from the TCC command, other options, and following file names by at least one space. You can also use a configuration file. See page 133 for details.

This chapter lists each of Turbo C++'s command-line compiler options in alphabetical order under option type, and describes
what each option does. The options are divided into three general types.

- compiler options
- linker options
- environment options

To see an onscreen list of the major command-line compiler options, type

```
tcc
```

at the DOS prompt (when you’re in the TURBOC directory or when TCC is in your DOS path), then press Enter.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>@filename</td>
<td>Response files</td>
</tr>
<tr>
<td>+filename</td>
<td>Tell TCC to use the alternate configuration file filename</td>
</tr>
<tr>
<td>-A</td>
<td>Use only ANSI keywords</td>
</tr>
<tr>
<td>-A- or -AT</td>
<td>Use Turbo C++ keywords (default)</td>
</tr>
<tr>
<td>-AK</td>
<td>Use only Kernighan and Ritchie keywords</td>
</tr>
<tr>
<td>-AU</td>
<td>Use only UNIX keywords</td>
</tr>
<tr>
<td>-a</td>
<td>Align word</td>
</tr>
<tr>
<td>-a-</td>
<td>Align byte (default)</td>
</tr>
<tr>
<td>-B</td>
<td>Compile and call the assembler to process inline assembly code</td>
</tr>
<tr>
<td>-b</td>
<td>Make enums word-sized (default)</td>
</tr>
<tr>
<td>-C</td>
<td>Nested comments on</td>
</tr>
<tr>
<td>-c</td>
<td>Compile to .OBJ but do not link</td>
</tr>
<tr>
<td>-Dname</td>
<td>Define name to the string consisting of the null character</td>
</tr>
<tr>
<td>-Dname=string</td>
<td>Defines name to string</td>
</tr>
<tr>
<td>-d</td>
<td>Merge duplicate strings on</td>
</tr>
<tr>
<td>-d-</td>
<td>Merge duplicate strings off (default)</td>
</tr>
<tr>
<td>-Efilename</td>
<td>Use filename as the assembler to use</td>
</tr>
<tr>
<td>-efilename</td>
<td>Link to produce filename.EXE</td>
</tr>
<tr>
<td>-f</td>
<td>Emulate floating point (default)</td>
</tr>
<tr>
<td>-f-</td>
<td>Don’t do floating point</td>
</tr>
<tr>
<td>-ff</td>
<td>Fast floating point (default)</td>
</tr>
<tr>
<td>-ff-</td>
<td>Strict ANSI floating point</td>
</tr>
<tr>
<td>-f87</td>
<td>Use 8087 hardware instructions</td>
</tr>
<tr>
<td>-f287</td>
<td>Use 80287 hardware instructions</td>
</tr>
<tr>
<td>-G</td>
<td>Optimize for speed</td>
</tr>
<tr>
<td>-G-</td>
<td>Optimize for size</td>
</tr>
<tr>
<td>-gn</td>
<td>Warnings: stop after n messages</td>
</tr>
<tr>
<td>-Ipathname</td>
<td>Directories for include files</td>
</tr>
<tr>
<td>-in</td>
<td>Make significant identifier length to be n</td>
</tr>
<tr>
<td>-jn-</td>
<td>Errors: stop after n messages</td>
</tr>
<tr>
<td>-K</td>
<td>Default character type unsigned</td>
</tr>
<tr>
<td>-K-</td>
<td>Default character type signed (default)</td>
</tr>
<tr>
<td>-k</td>
<td>Standard stack frame on (default)</td>
</tr>
</tbody>
</table>
Table 4.1: Command-line options summary (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-L pathname</td>
<td>Directories for libraries</td>
</tr>
<tr>
<td>-lx</td>
<td>Pass option x to the linker (can use more than one x)</td>
</tr>
<tr>
<td>-l-x</td>
<td>Suppress option x for the linker</td>
</tr>
<tr>
<td>-M</td>
<td>Instruct the linker to create a map file</td>
</tr>
<tr>
<td>-mc</td>
<td>Compile using compact memory model</td>
</tr>
<tr>
<td>-mh</td>
<td>Compile using huge memory model</td>
</tr>
<tr>
<td>-ml</td>
<td>Compile using large memory model</td>
</tr>
<tr>
<td>-mm</td>
<td>Compile using medium memory model</td>
</tr>
<tr>
<td>-mm!</td>
<td>Compile using medium model; assume DS != SS</td>
</tr>
<tr>
<td>-ms</td>
<td>Compile using small memory model (default)</td>
</tr>
<tr>
<td>-ms!</td>
<td>Compile using small model; assume DS != SS</td>
</tr>
<tr>
<td>-mt</td>
<td>Compile using tiny memory model</td>
</tr>
<tr>
<td>-mt!</td>
<td>Compile using tiny model; assume DS != SS</td>
</tr>
<tr>
<td>-N</td>
<td>Check for stack overflow</td>
</tr>
<tr>
<td>-n pathname</td>
<td>Output directory</td>
</tr>
<tr>
<td>-O</td>
<td>Optimize jumps</td>
</tr>
<tr>
<td>-O-</td>
<td>No optimization (default)</td>
</tr>
<tr>
<td>-ofilename</td>
<td>Compile source file to filename.obj</td>
</tr>
<tr>
<td>-P</td>
<td>Perform a C++ compile regardless of source file extension</td>
</tr>
<tr>
<td>-p</td>
<td>Use Pascal calling convention</td>
</tr>
<tr>
<td>-P-</td>
<td>Use C calling convention (default)</td>
</tr>
<tr>
<td>-Qe</td>
<td>Instructs the compiler to use all available EMS memory (default)</td>
</tr>
<tr>
<td>-Qe-</td>
<td>Instructs the compiler to not use any EMS memory (default)</td>
</tr>
<tr>
<td>-Qx</td>
<td>Instructs the compiler to use all available extended memory (default)</td>
</tr>
<tr>
<td>-Qx=nnnn</td>
<td>Instructs the compiler to reserve nnnn Kbytes of extended memory for other programs, and to use the rest itself</td>
</tr>
<tr>
<td>-Qx-</td>
<td>Instructs the compiler to not use any extended memory</td>
</tr>
<tr>
<td>-r</td>
<td>Use register variables on (default)</td>
</tr>
<tr>
<td>-r-</td>
<td>Suppresses the use of register variables.</td>
</tr>
<tr>
<td>-rd</td>
<td>Only allow declared register variables to be kept in registers</td>
</tr>
<tr>
<td>-S</td>
<td>Produce .ASM output file</td>
</tr>
<tr>
<td>-T string</td>
<td>Pass string as an option to TASM or assembler specified with -E</td>
</tr>
<tr>
<td>-T-</td>
<td>Remove all previous assembler options</td>
</tr>
<tr>
<td>-U name</td>
<td>Undefine any previous definitions of name</td>
</tr>
<tr>
<td>-u</td>
<td>Generate underbars on (default)</td>
</tr>
<tr>
<td>-v</td>
<td>Source debugging on</td>
</tr>
<tr>
<td>-vi</td>
<td>Controls expansion of inline functions</td>
</tr>
<tr>
<td>-w</td>
<td>Display warnings on</td>
</tr>
<tr>
<td>-w-</td>
<td>Display warnings off</td>
</tr>
<tr>
<td>-wxxx</td>
<td>Enable xxx warning message</td>
</tr>
<tr>
<td>-w-xxx</td>
<td>Disable xxx warning message</td>
</tr>
<tr>
<td>-X</td>
<td>Disable compiler autodependency output</td>
</tr>
</tbody>
</table>

For details about memory models, see Chapter 4 in the Programmer's Guide.
<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Y</td>
<td>Enable overlay code generation</td>
</tr>
<tr>
<td>-Yo</td>
<td>Overlay the compiled files</td>
</tr>
<tr>
<td>-y</td>
<td>Line numbers on</td>
</tr>
<tr>
<td>-Z</td>
<td>Enable register usage optimization</td>
</tr>
<tr>
<td>-zAname</td>
<td>Code class</td>
</tr>
<tr>
<td>-zBname</td>
<td>BSS class</td>
</tr>
<tr>
<td>-zCname</td>
<td>Code segment</td>
</tr>
<tr>
<td>-zDname</td>
<td>BSS segment</td>
</tr>
<tr>
<td>-zEname</td>
<td>Far segment</td>
</tr>
<tr>
<td>-zFname</td>
<td>Far class</td>
</tr>
<tr>
<td>-zGname</td>
<td>BSS group</td>
</tr>
<tr>
<td>-zHname</td>
<td>Far group</td>
</tr>
<tr>
<td>-zPname</td>
<td>Code group</td>
</tr>
<tr>
<td>-zRname</td>
<td>Data segment</td>
</tr>
<tr>
<td>-zSname</td>
<td>Data group</td>
</tr>
<tr>
<td>-zTname</td>
<td>Data class</td>
</tr>
<tr>
<td>-zX</td>
<td>Use default name for X. (default)</td>
</tr>
<tr>
<td>-1</td>
<td>Generate 80186 instructions</td>
</tr>
<tr>
<td>-1-</td>
<td>Generate 8088/8086 instructions and 80286 real-mode instructions (default)</td>
</tr>
<tr>
<td>-2</td>
<td>Generate 80286 protected-mode compatible instructions</td>
</tr>
</tbody>
</table>

### Turning options on and off

Use this feature to override settings in configuration files.

You select command-line options by entering 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 hyphen (−) turns the option off, and a plus sign (+) or nothing turns it on. So, for example, −C and −C+ both turn nested comments on, while −C− turns nested comments off.

### Syntax and file names

C++ files have the extension .CPP.

Turbo 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.C
filename.c    Compile FILENAME.C
filename.cpp   Compile FILENAMECPP
filename.xyz   Compile FILENAME.XYZ

For example, given the following command line

tcc -a -f -C -O -Z -emyexe oldfile1.c oldfile2 nextfile.c

TCC compiles OLDFILE1.C, OLDFILE2.C, and NEXTFILE.C to .OBJ, and linking produces an executable program file named MYEXE.EXE with word alignment (-a), floating-point emulation (-f), nested comments (-C), optimization (-O), and optimistic aliasing (-Z) selected.

TCC invokes TASM if you give it an .ASM file on the command line or if a .C file contains inline assembly. The options TCC gives to TASM are

/D_ _MODEL_ _ /D_ _lang_ _ /ml /fp

where MODEL is either TINY, SMALL, MEDIUM, COMPACT, LARGE, or HUGE. The /ml switch tells TASM to assemble with case sensitivity on. lang is CDECL or PASCAL; fp is r when you’ve specified -f87; e otherwise.

Response files

If you need to specify many options and/or files on the command line, you can place them in an ASCII text file. You can then tell TCC to read its command line from this file by including the appropriate file name prefixed with @ on the TCC command line. 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 SMALL.RSP contains CLOUDS.C and RAIN.C. The following TCC command will compile the files STARS.C, CLOUDS.C, RAIN.C, and MOON.C:

TCC STARS @SMALL.RSP MOON

Compiler options

Turbo C++’s command-line compiler options fall into nine groups. These groups are as follows:
1. **Memory model options** let you specify which memory model Turbo C++ will compile your program under. (The models are tiny, small, medium, compact, large, and huge.)

2. **Macro definitions** let you define macros (also known as manifest or symbolic constants) on the command line. The default definition is the null string. These options also let you undefine previously defined macros.

3. **Code generation options** govern characteristics of the generated code, such as the floating-point option, calling convention, character type, or CPU instructions.

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.

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.

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.

7. **Segment-naming control options** allows you to rename segments and to reassign their groups and classes.

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

9. **EMS and extended memory options** let you control how much expanded and extended memory Turbo C++ uses.

---

### Memory model

- `-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

The net effect of the new -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

-Dname Defines the named identifier name to the empty string.
-Dname=string Defines the named identifier name to the string string after the equal sign. string cannot contain any spaces or tabs.
-Uname Undefines any previous definitions of the named identifier name.

Turbo 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):
  tcc -Dxxx;yyy=1;zzz=NO myfile.c
- You can place more than one -D option on the command line:
  tcc -Dxxx -Dyyy=1 -Dzzz=NO myfile.c
- You can mix ganged and multiple -D listings:
  tcc -Dxxx -Dyyy=1;zzz=NO myfile.c

Code generation options

-1 Causes Turbo C++ to generate extended 80186 instructions. This option also generates 80286 programs
running in real mode, such as with the IBM PC/AT under DOS.

-2 Causes Turbo C++ to generate 80286 protected-mode compatible instructions.

-a 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 will be placed at an even-numbered address (off by default, allowing bytewise alignment).

-b Causes the compiler to always allocate a whole word for enum types. (That is also the way TC 2.0 treated enumerations.)

Normally, the compiler just allocates an unsigned or signed byte if the minimum and maximum values of the enumeration are both within 0 to 255 or -128 to 127, respectively.

-d Merges literal strings when one string matches another; this produces smaller programs (off by default).

-f Emulates 8087 calls at run time if the run-time system does not have an 8087; if it does have one, calls the 8087 for floating-point calculations (the default).

-f- Specifies that the program contains no floating-point calculations, so no floating-point libraries will be linked at the link step.

-ff Fast floating point. Compiler optimizes floating-point operations without regard to explicit or implicit type conversions. Answers can be faster than under ANSI operating mode. See Chapter 4, “Memory models, floating point, and overlays,” in the Programmer’s Guide for details.

-ff- Turns off the fast floating-point option. The compiler follows strict ANSI rules regarding floating-point conversions.

-f87 Generates floating-point operations using inline 80x87 instructions rather than using calls to 80x87 emulation library routines. Specifies that a math coprocessor will
be available at run time; programs compiled with this option will not run on a machine that does not have a math coprocessor.

-f287

Similar to -f87, but uses instructions that are only available with an 80287 (or higher) chip.

-K

Causes 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

Generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines. The default is on.

-N

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 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

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 usage, 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.

-u

With -u selected, when you declare an identifier, Turbo C++ automatically puts an underscore (_) in front before saving that identifier in the object module.

Turbo C++ treats Pascal-type identifiers (those modified by the pascal keyword) differently—they are uppercase and are not prefixed with an underscore.

Underscores for C identifiers are optional, but on by default. You can turn them off with -u-. However, if you are using the standard Turbo C++ libraries, you will encounter problems unless you rebuild the libraries. (To do this, you will need the Turbo C++
run-time library source code; contact Borland for more information.)

-\textbf{X}  
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 integrated environment. Normally this option is only used for files that are to be put into .LIB files (to save disk space).

-\textbf{Y}  
Generates overlay-compatible code. Every file in an overlaid program must be compiled with this option; see Chapter 4, “Memory models, floating point, and overlays,” in the \textit{Programmer’s Guide} for details on overlays.

-\textbf{Yo}  
Overlays the compiled file(s); see Chapter 4 in the \textit{Programmer’s Guide} for details.

-\textbf{y}  
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.

\textbf{Turbo Debugger is both a source level (symbolic) and assembly level debugger.}  

-\textbf{v}  
Tells the compiler to include debugging information in the .OBJ file so that the file(s) being compiled can be debugged with either Turbo C++’s integrated debugger or the standalone Turbo Debugger. The compiler also passes this switch on to the linker so it can include the debugging information in the .EXE file.

To facilitate debugging, this switch also causes C++ in-line functions to be treated as normal functions. If you want to avoid that, use -vi.

-\textbf{vi}  
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:

-\textbf{v}  
Turns debugging on and inline expansion off
-v- Turns debugging off and inline expansion on
-vi Turns inline expansion on
-vi- Turns inline expansion off

So, for example, if you want to turn both debugging and inline expansion on, you must use two switches:

-v vi

---

**Optimization options**

-\texttt{G} Causes the compiler to bias its optimization in favor of speed over size.

-\texttt{O} Turns optimizations on. This optimization eliminates redundant jumps (such as jumps to jumps) and multiple copies of identical code that jump to the same location (tail merging).

It also suppresses redundant register loads. When \texttt{Z} is not on, this will not change the behavior of your program (except, of course, that the code becomes more efficient).

-\texttt{-O} No optimization. Compiles the fastest, produces the poorest code.

-\texttt{r} Suppresses the use of register variables.

Unless you are an expert, don't use \texttt{r}.

When you are using the \texttt{-r} 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 \texttt{-r}.

On the other hand, if you are interfacing with existing assembly-language code that does not preserve SI,DI, the \texttt{-r} option allows you to call that code from Turbo C++.

-r Enables the use of register variables (the default).

-\texttt{rd} 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.

-\texttt{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 \texttt{-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 this assignment could potentially change any
variable. Therefore it has to forget about all copies of
variables in registers (i.e., 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

See Chapter 1, "The Turbo
C++ language standard," in
the Programmer's Guide for
a complete list of the Turbo
C++ keywords.

_compiles ANSI-compatible code: Any of the Turbo C++
extension keywords are ignored and can be used as
normal identifiers. These keywords include:

| _cs | _ss | far | near |
| _ds | asm | huge | pascal |
| _es | cdecl | interrupt |

and the register pseudovariables, such as _AX, _BX, _SI,
and so on.

-A  Use Turbo C++ keywords.
-AK Use only Kernighan and Ritchie keywords.
-AU Use only UNIX keywords.
-C  Allows nesting of comments. Comments may not
normally be nested.
-in Causes the compiler to recognize only the first n charac-
ters of identifiers. All identifiers, whether variables, pre-
processor macro names, or structure member names, are
treated as distinct only if their first n characters are
distinct.
By default, Turbo C++ uses 32 characters per identifier. Other systems, including UNIX, 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.

\[-P\] Compile as a C++ program, regardless of source file extension link with C++ libraries. If the command line doesn’t have any .CPP files, you’ll need this option to link in the C++ libraries.

Error-reporting options

The asterisk (*) indicates that the option is on by default. All others are off by default.

\[-gn\] Stops compiling after \(n\) messages (warning and error messages combined).

\[-Jn\] Stops compiling after \(n\) error messages.

\[-wxxx\] Enables the warning message indicated by \(xxx\). The option \[-w-xxx\] suppresses the warning message indicated by \(xxx\). See Chapter 7, “Error messages,” of the \textit{Programmer's Guide} for a detailed explanation of these warning messages. The possible options for \[-wxxx\] are listed here and divided into four categories: ANSI violations, frequent errors, portability warnings, and C++ warnings. You can also use the pragma \texttt{warn} in your source code to control these options. See the section on preprocessor directives in Chapter 1, “The Turbo C++ language standard,” in the \textit{Programmer's Guide}.

ANSI violations

\[-wbbf\] Bit fields must be \texttt{signed} or \texttt{unsigned int}.

\[-wbei*\] Initialization with inappropriate type.

\[-wbfs*\] Untyped bit field assumed \texttt{signed int}.

\[-wbig*\] Hexadecimal value contains more than three digits.

\[-wdcl*\] Declaration does not specify a tag or an identifier.

\[-wdpu*\] Declare \texttt{function} prior to use in prototype.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-wdup*</td>
<td>Redefinition of <code>macro</code> is not identical.</td>
</tr>
<tr>
<td>-weas</td>
<td>Assigning <code>integer_val</code> to <code>enumeration</code>.</td>
</tr>
<tr>
<td>-wext*</td>
<td><code>Identifier</code> is declared as both external and static.</td>
</tr>
<tr>
<td>-will</td>
<td>Ill-formed pragma.</td>
</tr>
<tr>
<td>-wpin</td>
<td>This initialization is only partially bracketed.</td>
</tr>
<tr>
<td>-wret*</td>
<td>Both return and return with a value used.</td>
</tr>
<tr>
<td>-wstr*</td>
<td>Functions may not be part of a struct or union.</td>
</tr>
<tr>
<td>-wstu*</td>
<td>Undefined structure <code>structure</code>.</td>
</tr>
<tr>
<td>-wsus*</td>
<td>Suspicious pointer conversion.</td>
</tr>
<tr>
<td>-wvoi*</td>
<td>Void functions may not return a value.</td>
</tr>
<tr>
<td>-wzdi*</td>
<td>Division by zero.</td>
</tr>
<tr>
<td>-wzst*</td>
<td>Zero length structure.</td>
</tr>
</tbody>
</table>

**Frequent errors**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-waus*</td>
<td><code>Identifier</code> is assigned a value that is never used.</td>
</tr>
<tr>
<td>-wdef*</td>
<td>Possible use of <code>identifier</code> before definition.</td>
</tr>
<tr>
<td>-weff*</td>
<td>Code has no effect.</td>
</tr>
<tr>
<td>-wpar*</td>
<td>Parameter <code>parameter</code> is never used.</td>
</tr>
<tr>
<td>-wplai</td>
<td>Possibly incorrect assignment.</td>
</tr>
<tr>
<td>-wrcx*</td>
<td>Unreachable code.</td>
</tr>
<tr>
<td>-wrvl</td>
<td>Function should return a value.</td>
</tr>
<tr>
<td>-wamb</td>
<td>Ambiguous operators need parentheses.</td>
</tr>
<tr>
<td>-wamp</td>
<td>Superfluous <code>&amp;</code> with function or array.</td>
</tr>
<tr>
<td>-wnod</td>
<td>No declaration for function <code>function</code>.</td>
</tr>
<tr>
<td>-wpro</td>
<td>Call to function with no prototype.</td>
</tr>
<tr>
<td>-wstv</td>
<td>Structure passed by value.</td>
</tr>
<tr>
<td>-wuse</td>
<td><code>Identifier</code> declared but never used.</td>
</tr>
</tbody>
</table>

**Portability warnings**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-wapt*</td>
<td>Nonportable pointer assignment.</td>
</tr>
<tr>
<td>-wcln</td>
<td>Constant is long.</td>
</tr>
<tr>
<td>-wcpt*</td>
<td>Nonportable pointer comparison.</td>
</tr>
<tr>
<td>-wrng*</td>
<td>Constant out of range in comparison.</td>
</tr>
<tr>
<td>-wrpt*</td>
<td>Nonportable return type conversion.</td>
</tr>
<tr>
<td>-wsig</td>
<td>Conversion may lose significant digits.</td>
</tr>
<tr>
<td>-wucp</td>
<td>Mixing pointers to <code>signed</code> and <code>unsigned char</code>.</td>
</tr>
</tbody>
</table>

**C++ warnings**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-watt</td>
<td>Assignment to <code>this</code> is obsolete; use <code>X::operator new</code> instead.</td>
</tr>
</tbody>
</table>
Program flow can skip this initialization, try using `{ }`.

Function1 hides virtual function function2.

Functions containing identifier are not expanded inline.

Temporary used to initialize identifier.

Temporary used for parameter in call to identifier.

Non-const function function called const object.

The constant member identifier is not initialized.

Base initialization without a class name is now obsolete.

This style of function definition is now obsolete.

Obsolete syntax, use :: instead.

Use of overload is now unnecessary and obsolete.

Identifier is both a structure tag and a name, now obsolete.

---

Segment-naming control

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.

Changes the name of the code segment class to name. By default, the code segment is assigned to class CODE.

Changes the name of the uninitialized data segment class to name. By default, the uninitialized data segments are assigned to class BSS.

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.)

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.

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.

Changes the name of the class for far objects to name. By default, the name is FAR_DATA.

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.
-zHname Causes far objects to be put into group name. By default, far objects are not put into a group.

-zPname Causes any output files to be generated with a code group for the code segment named name.

-zRname 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.

-zSname 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.

-zTname Sets the name of the initialized data segment class to name. By default the initialized data segment class is named DATA.

-zX* Uses the default name for X. For example, -zA* assigns the default class name CODE to the code segment.

Compilation control options

-B Compiles and calls the assembler to process inline assembly code.

-c Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link command.

-Ena me Uses name as the name of the assembler to use. By default, TASM is used.

-ofilename Compiles the named file to the specified filename.obj.

-P Compiles the source file as a C++ program regardless of its extension.

-S Compiles the named source files and produces assembly language output files (.ASM), but does not assemble. When you use this option, Turbo C++ will include the C source lines as comments in the produced .ASM file. This option is not available in the integrated environment.
EMS and extended memory options

If you are in doubt about your systems' overall use of extended memory, don't use this option.

-\texttt{Tstring} \hspace{1cm} Parses \textit{string} as an option to TASM (or as an option to the assembler defined with -\texttt{E}).

-\texttt{T-} \hspace{1cm} Removes all previously defined assembler options.

-\texttt{Qe} \hspace{1cm} Instructs the compiler to use all EMS memory it can find. This is on by default. This option speeds up your compilations, especially for large source files.

-\texttt{Qe-} \hspace{1cm} Instructs the compiler not to use any EMS memory.

-\texttt{Qx} \hspace{1cm} Instructs the compiler to use all extended memory it can find. Like -\texttt{Qe}, this speeds up compilations of large source files. However, unlike -\texttt{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.

-\texttt{Qx=nnnn} \hspace{1cm} Instructs the compiler to reserve \textit{nnnn} Kbytes 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 -\texttt{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.

-\texttt{Qx-} \hspace{1cm} Instructs the compiler not to use any extended memory. This is the default.

Linker options

\texttt{efilename} \hspace{1cm} Derives the executable program's name from \textit{filename} by adding the file extension .EXE (the
program name will then be filename.EXE). filename must immediately follow the \texttt{-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.

\textbf{\texttt{-M}} \quad Forces the linker to produce a full link map. The default is to produce no link map.

\textbf{\texttt{-Ix}} \quad Passes option \texttt{x} to the linker. The switch \texttt{-I-x} suppresses option \texttt{x}. More than one option can appear after the \texttt{-I}.

Environment options

\textbf{\texttt{-I directory}} \quad Searches \texttt{directory}, 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 (\texttt{:}). A directory is any valid directory or directory path. You can use more than one \texttt{-I} directory option.

\textbf{\texttt{-L directory}} \quad Forces the linker to get the C0x.OBJ start-up object file and the Turbo 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.

\textbf{\texttt{-nxxx}} \quad Places any .OBJ or .ASM files created by the compiler in the directory or drive named by the path \texttt{xxx}.

Turbo C++ can search multiple directories for include and library files. This means that the syntax for the library directories (\texttt{-L}) and include directories (\texttt{-I}) command-line options, like that of the \texttt{#define} option (\texttt{-D}), allows multiple listings of a given option.

Here is the syntax for these options:

\begin{itemize}
  \item \textbf{Library directories:} \quad \texttt{-Ldirname;dirname;...}
  \item \textbf{Include directories:} \quad \texttt{-Idirname;dirname;...}
\end{itemize}

The parameter \texttt{dirname} used with \texttt{-L} and \texttt{-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 -l option, separating ganged entries with a semicolon, like this:
  
  ```
tcc -Ldirname1;dirname2;dirname3 -linc1;inc2;inc3 myfile.c
  ```

- You can place more than one of each option on the command line, like this:
  
  ```
tcc -Ldirname1 -Ldirname2 -Ldirname3 -linc1 -linc2 -linc3 myfile.c
  ```

- You can mix ganged and multiple listings, like this:

  ```
tcc -Ldirname1;dirname2 -Ldirname3 -linc1;inc2 -linc3 myfile.c
  ```

If you list multiple -L or -l 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 integrated environment (TC.EXE) also supports multiple library directories, using the “gang entry” syntax.

---

**Library files**

Turbo C++ recognizes two types of library files: implicit and user-specified (also known as explicit library files).

- Implicit library files are the ones Turbo C++ automatically links in. These are the Cx.LIB files, CPx.LIB, EMU.LIB or FP87.LIB, MATHx.LIB, and the start-up object files (C0x.OBJ).

- User-specified library files are the ones you list on the command line or in a project file; these are file names with an .LIB extension.

---

**File-search algorithms**

The Turbo 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, Turbo 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, Turbo C++ searches for somefile.h first in the current directory; if it does not find the header file there, it
Your code written under any version of Turbo C will work without problems in Turbo C++.

then searches in the include directories specified in the command line.

The library file search algorithms are similar to those for include files:

- **Implicit libraries:** Turbo C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for `#include <somefile.h>`.

- **Explicit libraries:** Where Turbo C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name.

  - If you list an explicit library file name with no drive or directory (like this: `mylib.lib`), Turbo 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`), Turbo C++ searches only in the location you explicitly listed as part of the library path name and not in the specified library directories.

---

- **-L and -I and configuration files**

  The `-L` and `-I` options you list on the command line take priority over those in your configuration files.

---

**An example with notes**

Here is an example of using a TCC command line that incorporates multiple library directories (`-L`) and include directories (`-I`) options.

1. Your current drive is C:, and your current directory is `C:\TURBOC`, where TCC.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 Turbo 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 TCC command line:

tcc -mm -Llib;starlib -Iinclude orion umaj parx.lib a:\astrolib\warp.lib

TCC compiles ORION.C and UMAJ.C to .OBJ files.

The compiler searches C:\TURBOC\INCLUDE for the include files in your source code, then links them with the medium model start-up code (COM.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 (search ends 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.

The TURBOC.CFG File

You can set up a list of options in a configuration file called TURBOC.CFG, which can be used in addition to options entered on the command line. This configuration file contains options as they would be entered on the command line.

If you’ve listed your commonly used options in TURBOC.CFG, you won’t need to enter them on the command line when you use...
TCC.EXE. If you don’t want to use certain options that are listed in TURBOC.CFG, you can override them with switches on the command line.

You can create the TURBOC.CFG file using any standard ASCII editor or word processor (such as Turbo C++’s integrated editor). You can list options (separated by spaces) on the same line or list them on separate lines. Then, when you compile your program from the command line, Turbo C++ uses the options supplied in TURBOC.CFG, in addition to the ones given on the command line.

When you run TCC, it 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, Turbo C++ then looks in the start directory (where TCC.EXE resides). Note that TURBOC.CFG is not the same as TCCONFIG.TC, which is the default integrated environment version of a configuration file.

Options given on the command line override the same options specified in TURBOC.CFG. 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.

How are command-line options and TURBOC.CFG options combined and overridden? There are two kinds of TURBOC.CFG options:

- the -I and -L options
- all other options in the file

Under any circumstances, 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 switch on the right cancels an on switch to the left.)
- The -I and -L options on the left, however, take precedence over those on the right.

When the options from the configuration file are combined with the command-line options, the -I and -L options from TURBOC.CFG are appended to the right of the command-line options, and the remaining TURBOC.CFG options are inserted on
the left of the command line's list of options, immediately after the TCC command.

Thus, because of the way the command line and TURBOC.CFG are combined, the TURBOC.CFG -I and -L options are on the extreme right, so the include and library directories specified in the command line are the first ones that Turbo C++ searches for the include and library files. This gives the -I and -L directories on the command line priority over those in the configuration file. All other options from the TURBOC.CFG file are inserted to the left of the command-line options, which again, correctly, gives the command-line options priority over them.

Using an alternate configuration file

You can tell TCC to read options from a file other than the default TURBOC.CFG. To specify the alternate configuration file name, include its file name, prefixed with +, anywhere on the TCC command line.

For example, to read the option settings from the file D:\ALT.CFG, you could use the following command line:

    TCC +D:\ALT.CFG ......

TCCNVT.EXE takes a configuration file created by one environment (the integrated environment or the command-line compiler) and converts it for use by the other.

The conversion command is

    TCCNVT SourceFile [DestinationFile]

TCCNVT automatically determines the direction of the conversion: It examines the source file to see whether it is an integrated environment configuration file or a command-line compiler configuration file.

The destination file name is optional. If you don’t specify a file name, TCCNVT uses the default name TCCNVT.TC or TURBOC.CFG, depending on the conversion direction. You can give any file name.

When it creates the TCCNVT.TC file, TCCNVT uses default values for any items not specified by the command-line compiler configuration file (TURBOC.CFG). Going in the other direction, it
includes in TURBOC.CFG only the options in TCCNVT.TC that
differ from the default values.

TCCNVT returns you to the DOS prompt when the conversion is
done.
Your Turbo C++ package supplies much more than just two versions of the fastest C compiler available. It also provides eleven powerful standalone utilities that you can use with your Turbo C++ files or your other modules. Most of these utilities are documented in a text file included with your distribution disks.

These highly useful adjuncts to Turbo C++ are

- BGIOBJ (a conversion utility for graphics drivers and fonts)
- CINSTXFR (an integrated environment transfer utility)
- CPP (the preprocessor)
- GREP (a file-search utility)

Documented herein.

- MAKE (the standalone program manager)
- OBJXREF (an object module cross-referencer)
- PRJCVT (converts Turbo C project files to the Turbo C++ format)

Documented herein.

- PRJ2MAK (converts Turbo C++ project files to MAKE files)
- THELP (the Turbo Help utility)

Documented herein.

- TLIB (the Turbo Librarian)
- TLINK (the Turbo Linker)
- TOUCH (the file date and time changer)
- TRIGRAPR (a character-conversion utility)
This chapter explains what MAKE, TLIB, TLINK, and TOUCH do, and illustrates, with code and command-line examples, how to use them.

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.

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 timestamp 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:

- 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, re-compiling 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 159 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 161 for more on this MAKE directive) in the current directory. If you use the –I
MAKE

(include) option, it will also search in the directory specified with the \texttt{-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 \texttt{-d} is not a valid substitution for \texttt{-D}.

<table>
<thead>
<tr>
<th>Option</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{-?} or \texttt{-h}</td>
<td>Prints a help message. The default options are displayed with plus signs following. (This makes them default.)</td>
</tr>
<tr>
<td>\texttt{-a}</td>
<td>Causes an automatic dependency check on .OBJ files.</td>
</tr>
<tr>
<td>\texttt{-B}</td>
<td>Builds all targets regardless of file dates.</td>
</tr>
<tr>
<td>\texttt{-Didentifier}</td>
<td>Defines the named identifier to the string consisting of the single character 1 (one).</td>
</tr>
<tr>
<td>\texttt{-Diden=string}</td>
<td>Defines the named identifier \texttt{iden} to the string after the equal sign. The string cannot contain any spaces or tabs.</td>
</tr>
<tr>
<td>\texttt{-ffilename}</td>
<td>Uses \texttt{filename} as the MAKE file. If \texttt{filename} does not exist and no extension is given, tries \texttt{FILENAME.MAK}.</td>
</tr>
<tr>
<td>\texttt{-i}</td>
<td>Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. This is equivalent to putting \texttt{-} in front of all commands in the MAKEFILE (described below).</td>
</tr>
<tr>
<td>\texttt{-idirectory}</td>
<td>Searches for include files in the indicated directory (as well as in the current directory).</td>
</tr>
<tr>
<td>\texttt{-K}</td>
<td>Keeps (does not erase) temporary files created by MAKE. All temporary files have the form \texttt{MAKE\texttt{nnnn}.$$}, where \texttt{nnnn} ranges from 0000 to 9999. See page 146 for more on temporary files.</td>
</tr>
<tr>
<td>\texttt{-n}</td>
<td>Prints the commands but does not actually perform them. This is useful for debugging a makefile.</td>
</tr>
<tr>
<td>\texttt{-s}</td>
<td>Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed.</td>
</tr>
<tr>
<td>\texttt{-S}</td>
<td>Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules.</td>
</tr>
<tr>
<td>\texttt{-Uidentifier}</td>
<td>Undefines any previous definitions of the named identifier.</td>
</tr>
<tr>
<td>\texttt{-W}</td>
<td>Writes the current specified non-string options (like \texttt{-s} and \texttt{-a}) to MAKE.EXE.</td>
</tr>
</tbody>
</table>

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
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.

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:

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
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 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.
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 Turbo C++'s built-in editor, Sprint, MicroStar, 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

file. This rule, and others following, make use of a macro. See page 155 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.
• 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:

```
# 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>
<thead>
<tr>
<th>Prefix</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>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.</td>
</tr>
<tr>
<td>--num</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>-4 myprog sample.x</td>
</tr>
<tr>
<td></td>
<td>If no --num prefix is given and the status is nonzero, MAKE stops and deletes the current target file.</td>
</tr>
<tr>
<td>--</td>
<td>With a hyphen but no number, MAKE will not check the exit status at all. Regardless of the exit status, MAKE continues.</td>
</tr>
</tbody>
</table>

Command body

The command body is treated exactly as if it were entered as a line to COMMAND.COM, 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:

```
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 Turbo C++ for processing. MAKE could run TCC.EXE once for each file, but it’s much more efficient to invoke TCC.EXE with a list of all the files to be compiled on the command line. This saves the overhead of reloading Turbo 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:

```
TCC {file1.c}
TCC {file2.c}
TCC {file3.c}
```

Rather than invoking Turbo C++ three times, MAKE issues the single command

```
TCC 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:
    TCC -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 TCC
- 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 TCC (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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>del</td>
<td>path</td>
<td>set</td>
</tr>
<tr>
<td>cd</td>
<td>dir</td>
<td>prompt</td>
<td>time</td>
</tr>
<tr>
<td>chdir</td>
<td>echo</td>
<td>rd</td>
<td>type</td>
</tr>
<tr>
<td>cls</td>
<td>erase</td>
<td>rem</td>
<td>ver</td>
</tr>
<tr>
<td>copy</td>
<td>for</td>
<td>ren</td>
<td>verify</td>
</tr>
<tr>
<td>ctty</td>
<td>md</td>
<td>rename</td>
<td>vol</td>
</tr>
<tr>
<td>date</td>
<td>mkdir</td>
<td>rmdir</td>
<td></td>
</tr>
</tbody>
</table>

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 145 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,

```plaintext
.c.obj
  tcc -c $<
prog.obj:

prog.obj depends on prog.c; it will execute the command line

  TCC -c prog.c

if out of date.
```

**Examples**

Here are some examples of explicit rules:

1. **prog.exe:** myprog.obj prog2.obj
   
   ```plaintext
   tcc myprog.obj prog2.obj
   ```

2. **myprog.obj:** myprog.c include\stdio.h
   
   ```plaintext
   tcc -c myprog.c
   ```

3. **prog2.obj:** prog2.c include\stdio.h
   
   ```plaintext
   tcc -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
MAKE

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

Turbo C++ works with MAKE to provide automatic dependency
checking for include files. TCC and TC 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. 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
tcc -c myprog.c

Now assume that the following source file, called MYPROG.C,
has been compiled with TCC (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 TCC. 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:
tcc -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 155. 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:
tcc -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

```
tcc -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:
tcc -c $<
```

You could then remove the command from the rule:

```
myprog.obj: myprog.c include\stdio.h
tcc -c myprog.c
```

and it would execute exactly as before.

If you're using Turbo 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:
tcc -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 TCC commands will use the switch `-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:

```
tcc -c -mm${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

```
tcc -c -mm myprog.c
```

Now, changing memory models is easy. If you change the first line to

```
MODEL = 1
```

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
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.

**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 in hexadecimal (for this version, 0x0300).

<table>
<thead>
<tr>
<th>Macro</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$d</code></td>
<td>Defined test macro</td>
</tr>
<tr>
<td><code>$*</code></td>
<td>Base file name macro with path</td>
</tr>
<tr>
<td><code>$&lt;</code></td>
<td>Full file name macro with path</td>
</tr>
<tr>
<td><code>$:</code></td>
<td>Path only macro</td>
</tr>
<tr>
<td><code>$.$</code></td>
<td>Full file name macro, no path</td>
</tr>
<tr>
<td><code>$&amp;</code></td>
<td>Base file name macro, no path</td>
</tr>
</tbody>
</table>

**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:

```make
!if !$d(MODEL)  
  MODEL=m
!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:
tcc -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
tcc -c $*
```
copies MYLIB.OBJ to the directory \OLDOBJ before compiling MYLIB.C.

In an implicit rule, $<$ takes on the file name plus the source extension. For example, the implicit rule

```
.c.obj:
tcc -c *c
```

produces exactly the same result as

```
.c.obj:
tcc -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\n
```

**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>
<thead>
<tr>
<th>MAKE directives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.autodepend</td>
<td>Turns on autodependency checking.</td>
</tr>
<tr>
<td>.elif</td>
<td>Conditional execution.</td>
</tr>
<tr>
<td>.else</td>
<td>Conditional execution.</td>
</tr>
<tr>
<td>.endif</td>
<td>Conditional execution.</td>
</tr>
<tr>
<td>.error</td>
<td>Causes MAKE to stop and print an error message.</td>
</tr>
<tr>
<td>.if</td>
<td>Conditional execution.</td>
</tr>
<tr>
<td>.ignore</td>
<td>Tells MAKE to ignore return value of a command.</td>
</tr>
</tbody>
</table>
Table 5.1: MAKE directives (continued)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include</code></td>
<td>Specifies a file to include in the makefile.</td>
</tr>
<tr>
<td><code>.noautodepend</code></td>
<td>Turns off autodependency checking.</td>
</tr>
<tr>
<td><code>.noignore</code></td>
<td>Turns off <code>.ignore</code>.</td>
</tr>
<tr>
<td><code>.nosilent</code></td>
<td>Tells MAKE to print commands before executing</td>
</tr>
<tr>
<td></td>
<td>them.</td>
</tr>
<tr>
<td><code>.noswap</code></td>
<td>Tells MAKE to not swap itself in and out of</td>
</tr>
<tr>
<td></td>
<td>memory.</td>
</tr>
<tr>
<td><code>.path.ext</code></td>
<td>Gives MAKE a path to search for files with</td>
</tr>
<tr>
<td></td>
<td>extension <code>.EXT</code>.</td>
</tr>
<tr>
<td><code>.silent</code></td>
<td>Tells MAKE to not print commands before</td>
</tr>
<tr>
<td></td>
<td>executing them.</td>
</tr>
<tr>
<td><code>.swap</code></td>
<td>Tells MAKE to swap itself in and out of memory.</td>
</tr>
<tr>
<td><code>.undef</code></td>
<td>Causes the definition for a specified macro to</td>
</tr>
<tr>
<td></td>
<td>be forgotten.</td>
</tr>
</tbody>
</table>

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.extension`

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:
```

Turbo C++ User's Guide
```
tcc -c $*
tmp.exe: tmp.obj
tcc 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:
   tcc -c -o$(.PATH.obj)$& $<
.obj.exe:
   tcc -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 (\texttt{!if}, \texttt{!elif}, \texttt{!else}, and \texttt{!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 \texttt{-D} option) can enable or disable sections of the makefile.

The format of these directives parallels those in C, assembly language, and Turbo Pascal:

\begin{verbatim}
!if expression
[ lines ]
!endif

!if expression
[ lines ]
!else
[ lines ]
!endif

!if expression
[ lines ]
!elif expression
[ lines ]
!endif
\end{verbatim}

\textit{Note} \textit{[lines]} can be any of the following statement types:

- \texttt{macro\_definition}
- \texttt{explicit\_rule}
- \texttt{implicit\_rule}
- \texttt{include\_directive}
- \texttt{if\_group}
- \texttt{error\_directive}
- \texttt{undef\_directive}

The conditional directives form a group, with at least an \texttt{!if} directive beginning the group and an \texttt{!endif} directive closing the group.

- One \texttt{!else} directive can appear in the group.
- \texttt{!elif} directives can appear between the \texttt{!if} and any \texttt{!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 \( \text{if} \) directives must have matching \( \text{lendif} \) 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 \( \text{lendif} \) directive:

\[
\begin{verbatim}
\text{if } $(\text{FILE\_COUNT}) > 5
  \text{some rules}
\text{else}
  \text{other rules}
<end-of-file>
\end{verbatim}
\]

**Expressions allowed in conditional directives**

Expressions are allowed in an \( \text{if} \) or an \( \text{elif} \) 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:

\[
\begin{align*}
4536 & \quad \text{# decimal constant} \\
0677 & \quad \text{# octal constant (distinguished by leading 0)} \\
0x23af & \quad \text{# hexadecimal constant (distinguished by leading Ox)}
\end{align*}
\]

An expression can use any of the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Unary operators</strong></td>
</tr>
<tr>
<td>–</td>
<td>Negation (unary minus)</td>
</tr>
<tr>
<td>~</td>
<td>Bit complement (inverts all bits)</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT (yields 0 if operand is nonzero, 1 otherwise)</td>
</tr>
<tr>
<td></td>
<td><strong>Binary operators</strong></td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>–</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
</tr>
</tbody>
</table>
&& Logical AND
|| Logical OR
>
Greater than
<
Less than
>=
Greater than or equal
<=
Less than or equal
== Equality
!= Inequality

**Ternary operator**

? : The operand before the ? is treated as a test.

If the value of the first operand is nonzero, then the second operand (the part between the ? and :) is the result.

If the value of the first operand is zero, the value of the result is the value of the third operand (the part after the :).

Parentheses can be used to group operands in an expression. In the absence of parentheses, all the unary operators take precedence over binary operators. The binary operators have the same precedences as they do in the C language, and are listed here in order of decreasing precedence.

*  /  %  Multiplicative operators
+  -  Additive operators
<<  >> Bitwise shift operators
<=  >= Relational operators
<  > Relational operators
==  =  != Relational operators
&  Bitwise AND
^  Bitwise exclusive OR
|  Bitwise OR
&& Logical AND
|| Logical OR

Operators of equal precedence are executed from left to right, except for nested ternary operators (?:), which are executed right to left.

Since this many layers of operator precedence can be confusing even to C experts, we recommend that you use parentheses liberally in your expressions.
You can invoke macros within an expression; the special macro $d(\cdot)$ 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 undefined directive**

The macro "undefined" 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 error messages**

MAKE diagnostic messages fall into two classes: errors and fatal errors.

- When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart the compilation.
- Errors indicate some sort of syntax or semantic error in the source makefile.

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.
The error 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

In order to look this error message up, 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 messages

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.

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:

filea: fileb
fileb: filec
filec: filea

This implies that filea depends on fileb, which depends on filec, and filec depends on filea. This is illegal, since a file cannot depend on itself, indirectly or directly.
Error directive: message
MAKE has processed an #error directive in the source file, and the text of the directive is displayed in the message.

Incorrect command-line argument: argument
You’ve used incorrect command-line arguments.

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.

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.

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.

Unable to open makefile
The current directory does not contain a file named MAKEFILE, and there is no MAKEFILE.MAK.

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.

Errors

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.

Bad undef statement syntax
An !undef statement must contain a single identifier and nothing else as the body of the statement.

Character constant too long
Character constants can be only one or two characters long.
Command arguments too long
The arguments to a command were more than the 127-character limit imposed by DOS.

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.

Command too long
The length of a command has exceeded 128 characters. You might wish to use a response file.

Division by zero
A divide or remainder in an if statement has a zero divisor.

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.

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.

If statement too long
An if statement has exceeded 4,096 characters.

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.

Illegal octal digit
An octal constant was found containing a digit of 8 or 9.

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.
Misplaced elif statement
An !elif directive is missing a matching !if directive.

Misplaced else statement
There's an !else directive without any matching !if directive.

Misplaced endif statement
There's an !endif directive without any matching !if directive.

No file name ending
The file name in an include statement is missing the
 correct closing quote or angle bracket.

Redefinition of target filename
The named file occurs on the left side of more than one explicit rule.

Rule line too long
An implicit or explicit rule was longer than 4,096 characters.

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.

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.

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 Turbo C++ were built with TLIB. You can use TLIB to build your own libraries, or to modify the Turbo 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 174) for details.

Although TLIB is not essential to creating executable programs with Turbo C++, it is a useful programmer's 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
The TUB linker, because it only opens a single file, instead of one file for each object module.

---

**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] [/Psize] [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 176.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tlib</strong></td>
<td>The command name that invokes TLIB.</td>
</tr>
<tr>
<td><strong>libname</strong></td>
<td>The DOS path name of the library you want to create or manage. Every TLIB command must be given a <code>libname</code>. Wildcards are not allowed. TLIB assumes an extension of <code>.LIB</code> if none is given. We recommend that you do not use an extension other than <code>.LIB</code>, since both TCC and TC’s project-make facility require the <code>.LIB</code> extension in order to recognize library files. <strong>Note:</strong> If the named library does not exist and there are <code>add</code> operations, TLIB creates the library.</td>
</tr>
<tr>
<td><strong>/C</strong></td>
<td>The case-sensitive flag. This option is not normally used; see page 175 for a detailed explanation.</td>
</tr>
<tr>
<td><strong>/E</strong></td>
<td>Create Extended Dictionary; see page 174 for a detailed explanation.</td>
</tr>
<tr>
<td><strong>/Psize</strong></td>
<td>Set the library page size to <code>size</code>; see page 174 for a detailed explanation.</td>
</tr>
<tr>
<td><strong>operations</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>listfile</strong></td>
<td>The name of the file listing library contents. The <code>listfile</code> 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 <code>listfile</code> is <code>.LST</code>. You can direct the listing to the screen by using the <code>listfile</code> name CON, or to the printer by using the name PRN.</td>
</tr>
</tbody>
</table>
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:
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.

<table>
<thead>
<tr>
<th>Action symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td>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.</td>
</tr>
<tr>
<td>-</td>
<td>Remove</td>
<td>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.</td>
</tr>
<tr>
<td>*</td>
<td>Extract</td>
<td>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.</td>
</tr>
<tr>
<td>-*</td>
<td>Extract &amp; Remove</td>
<td>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.</td>
</tr>
<tr>
<td>*-</td>
<td>Remove</td>
<td>TLIB replaces the named module with the corresponding file. This is just a shorthand for a remove followed by an add operation.</td>
</tr>
</tbody>
</table>

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 Turbo 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.
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
```

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.

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 \(\text{lookup}\) and \(\text{LOOKUP}\) 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.

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 \(\text{stars, Stars, and STARS}\) 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, Turbo C++ would treat \(\text{stars, Stars, and STARS}\) as three separate identifiers. As long as you use the library only with TLINK, you can use the TLIB \(/C\) option without any problems.

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.
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 new version of TLINK has more features, handles much larger programs, and is still quite fast.

The IDE has its own built-in linker. For the command-line version of Turbo C++, the linker, TLINK, 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 as a standalone linker.
By default, TCC 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 that looks like this:

```
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
Syntax: TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m • map file with publics
        /x • no map file at all
        /i • initialize all segments
        /l • include source line numbers
        /s • detailed map of segments
        /n • no default libraries
        /d • warn if duplicate symbols in libraries
        /c • lower case significant in symbols
        /3 • enable 32-bit processing
        /v • include full symbolic debug information
        /e • ignore Extended Dictionary
        /t • generate COM file
        /o • overlay switch
        /ye • expanded memory swapping
        /yx • extended memory swapping
```

In TLINK’s summary display, the line

```
Syntax: TLINK objfiles, exefile, mapfile, libfiles
```

specifies that you supply file names in the given order, separating the file types with commas.

For example, if you supply the command line

```
tlink /c mainline wd ln tx,fin,mfin,lib\comm 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 LIB.

TLINK appends extensions to file names that have none:
TLINK

- .OBJ for object files
- .EXE for executable files
- .MAP for map files
- .LIB for library files

If no .EXE file name is specified, TLINK derives the name of the executable file by appending .EXE to the first object file name listed. If, for example, you had not specified FIN as the .EXE file name in the previous example, TLINK would have created MAINLINE.EXE as your executable file.

When you use the /t option, the executable file extension defaults to .COM rather than .EXE.

TLINK always generates a map file, unless you explicitly direct it not to by including the /x option on the command line.

- If you give the /m option, the map file will include a list of public symbols.
- If you give the /s option, the map file will include a detailed segment map.

These are the rules TLINK follows when determining the name of the map file.

- If you don't specify any .MAP files, TLINK derives the map file name by adding a .MAP extension to the .EXE file name. (You can give the .EXE file name on the command line or in the response file; if no .EXE name is given, TLINK will derive it from the name of the first .OBJ file.)
- If you specify a map file name in the command line (or in the response file), TLINK adds the .MAP extension to the given name.

Even if you specify a map file name, if you use the /x option, TLINK won't create any map files at all.

Using response files

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/or 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.

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 previous command-line example as a response file, FINRESP, like this:

```
/c mainline wd+
  ln tx,fin
  mfin
  lib\comm 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:

<table>
<thead>
<tr>
<th>File name</th>
<th>Contents</th>
</tr>
</thead>
</table>
| LISTOBS   | mainline+
            | wd+
            | ln tx
             | lib\comm+
             | lib\support |
| LISTLIBS  |         |

You would then enter the TLINK command as
```
tlink /c @listobs,fin,mfin,@listlibs
```

Turbo C++ supports six different memory models: tiny, small, compact, medium, large, and huge. When you create an executable Turbo C++ file using TLINK, you must include the initialization module and libraries for the memory model being used.

The general format for linking Turbo C++ programs with TLINK is (be sure to include a path for the startup code and libraries)
```
tlink C0x myobjs, exe,[map],[mylibs] [overlay] [emu | fp87 mathx] Cx
```

where these file names represent the following:
TLINK

myobjs = the .OBJ files you want linked
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)

The other file names on this general TLINK command line represent Turbo C++ files, as follows:

C0x = initialization module for memory model t, s, c, m, l, or h
emu | fp87 = the floating-point libraries (choose one)
mathx = math library for memory model s, c, m, l, or h
Cx = run-time library for memory model s, c, m, l, or h
overlay = overlay manager library; needed only for overlaid programs

Startup code

The initialization modules have the name C0x.OBJ, where x is a single letter corresponding to the model: t, s, c, m, l, or h. Failure to link in the appropriate initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved and/or that no stack has been created.

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 .EXE file name on the TLINK command line. Otherwise, your program name will be C0x.EXE—probably not what you wanted!

Libraries

The order of objects and libraries is very important. You must always put the C start-up module (C0x.OBJ) first in the list of objects. Then, the library list should contain, in this specific order:

- your own libraries (if any)
- FP87.LIB or EMU.LIB, followed by MATHx.LIB (only necessary if you are using floating point)
- Cx.LIB (standard Turbo C++ run-time library file)
If you are using any Turbo C++ graphics functions, you must link in GRAPHICS.LIB. The graphics library is independent of memory models.

If you want to overlay your program, you must include OVERLAY.LIB; this library must precede the Cx.LIB library.

If your program uses any floating-point, you must include a floating-point library (EMU.LIB or FP87.LIB) plus a math library (MATHx.LIB) in the link command. Turbo C++'s two floating-point libraries are independent of the program's memory model.

- If you want to include floating-point emulation logic so that the program will work on machines whether they have a math coprocessor (80x87) chip or not, you must use EMU.LIB.
- 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 MATHx.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 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 Cx.LIB, where x is a single letter corresponding to the model, as before.

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:

- Tiny: C0T.OBJ, MATHS.LIB, CS.LIB
- Small: C0S.OBJ, MATHS.LIB, CS.LIB
- Compact: C0C.OBJ, MATHC.LIB, CC.LIB
- Medium: C0M.OBJ, MATHM.LIB, CM.LIB
- Large: C0L.OBJ, MATHL.LIB, CL.LIB
- Huge: C0H.OBJ, MATHH.LIB, CH.LIB

Note that the tiny and small models use the same libraries, but have different startup files (C0T.OBJ vs. C0S.OBJ).
Using TLINK with TCC

You can also use TCC, the standalone Turbo C++ compiler, as a “front end” to TLINK that will invoke TLINK with the correct startup file, libraries, and executable program name.

To do this, you give file names on the TCC command line with explicit .OBJ and .LIB extensions. For example, given the following TCC command line,

```
tcc -mx mainfile.obj sub1.obj mylib.lib
```

TCC will invoke TLINK with the files C0x.OBJ, EMU.LIB, MATHx.LIB and Cx.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.

When TCC 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-specifying character (m, x, i, l, s, n, d, c, 3, v, e, o, t, ye, or yx). (The DOS switch character is / by default. You can change it by using an INT 21H call.)

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.

/x, /m, /s 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 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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Length (Bytes)</th>
<th>Class</th>
<th>Segment Name</th>
<th>Group</th>
<th>Module</th>
<th>Alignment/Combining</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0000</td>
<td>055B</td>
<td>C•CODE</td>
<td>S•SYMB.Text</td>
<td>(none)</td>
<td>M•SYMB.C</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>00E5:000B</td>
<td>2735</td>
<td>C•CODE</td>
<td>S•QUAL.Text</td>
<td>(none)</td>
<td>M•QUAL.C</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>0359:0000</td>
<td>002B</td>
<td>C•CODE</td>
<td>S•SCOPY.Text</td>
<td>(none)</td>
<td>M•SCOPY</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>0358:000B</td>
<td>003A</td>
<td>C•CODE</td>
<td>S•LRSH.Text</td>
<td>(none)</td>
<td>M•LRSH</td>
<td>ACBP=20</td>
</tr>
<tr>
<td>035F:0005</td>
<td>00B3</td>
<td>C•CODE</td>
<td>S•PADA.Text</td>
<td>(none)</td>
<td>M•PADA</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>0367:000B</td>
<td>005B</td>
<td>C•CODE</td>
<td>S•PADD.Text</td>
<td>(none)</td>
<td>M•PADD</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>0360:0003</td>
<td>0025</td>
<td>C•CODE</td>
<td>S•POBP.Text</td>
<td>(none)</td>
<td>M•POBP</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>036F:0008</td>
<td>05CE</td>
<td>C•CODE</td>
<td>S•BRK.Text</td>
<td>(none)</td>
<td>M•BRK</td>
<td>ACBP=2B</td>
</tr>
<tr>
<td>03CC:0006</td>
<td>066F</td>
<td>C•CODE</td>
<td>S•FLOAT.Text</td>
<td>(none)</td>
<td>M•FLOAT</td>
<td>ACBP=20</td>
</tr>
<tr>
<td>0433:0006</td>
<td>000B</td>
<td>C•DATA</td>
<td>S•DATA</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
<tr>
<td>0433:0012</td>
<td>0003</td>
<td>C•DATA</td>
<td>S•DATA</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
<tr>
<td>0433:00E6</td>
<td>000E</td>
<td>C•DATA</td>
<td>S•DATA</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
<tr>
<td>0442:0004</td>
<td>0004</td>
<td>C•BSS</td>
<td>S•BSS</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
<tr>
<td>0442:0008</td>
<td>0002</td>
<td>C•BSS</td>
<td>S•BSS</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
<tr>
<td>0442:000A</td>
<td>000E</td>
<td>C•BSS</td>
<td>S•BSS</td>
<td>D•GROUP</td>
<td>M•SYMB.C</td>
<td>ACBP=4B</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The A field (alignment)</td>
<td>00</td>
<td>An absolute segment.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>A byte-aligned segment.</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>A word-aligned segment.</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>A paragraph-aligned segment.</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>A page-aligned segment.</td>
</tr>
<tr>
<td></td>
<td>A0</td>
<td>An unnamed absolute portion of storage.</td>
</tr>
<tr>
<td>The C field (combination)</td>
<td>00</td>
<td>May not be combined.</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>A public combining segment.</td>
</tr>
<tr>
<td>The B field (big)</td>
<td>00</td>
<td>Segment less than 64K.</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>Segment exactly 64K.</td>
</tr>
</tbody>
</table>
When a detailed map is requested through use of the /s switch, 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:

```
OC7F:031E idle Symbol1
0000:3EA2 Symbol2
OC7F:0320 idle Symbol3
```

/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 tell TLINK to create no map at all (using the /x option), this option will have no effect.

/l (uninitialized trailing segments) The /l 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.

/n (ignore default libraries) The /n option causes the linker to ignore default libraries specified by some compilers. You'll need this option if the default libraries are in another directory, because TLINK does not support searching for libraries. You may want to use this option when linking modules written in another language.

/c (case sensitivity) The /c option forces the case to be significant in public and external symbols. For example, by default, TLINK regards cloud, Cloud, and CLOUD as equal; the /c option makes them different.

/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 Turbo 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 (extended dictionary) The library files that are shipped with Turbo 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 169). The /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 switch if you have a program that needs slightly more memory to link when an extended dictionary is used.

Unless you use /e, TLINK will ignore any debugging information contained in a library that has an extended dictionary.

/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.

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.

/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 for all modules that contain 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. 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.

/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.

/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 4, "Memory models, floating point, and overlays," 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>
<thead>
<tr>
<th>Option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>Overlay all code segments until next comma or o−.</td>
</tr>
<tr>
<td>o−</td>
<td>Stop overlaying.</td>
</tr>
<tr>
<td>Switch</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>/OVY</code></td>
<td>Overlay segments of class OVY until the next comma or <code>O-</code>.</td>
</tr>
<tr>
<td><code>/CODE OOVLY</code></td>
<td>Overlay segments of class CODE or class OVL Y until next comma or <code>O-</code>.</td>
</tr>
<tr>
<td><code>/#FO</code></td>
<td>Use interrupt vector 0F0H for overlays.</td>
</tr>
</tbody>
</table>

You can use the `O` option in response files. If you use the `O` option in a response file, 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.

This switch 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.

This switch has several forms, shown below

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/ye</code></td>
<td>enable expanded memory swapping (default)</td>
</tr>
<tr>
<td><code>/ye-</code></td>
<td>disable expanded memory swapping</td>
</tr>
<tr>
<td><code>/yx</code></td>
<td>enable extended memory swapping</td>
</tr>
<tr>
<td><code>/yx-</code></td>
<td>disable extended memory swapping (default)</td>
</tr>
</tbody>
</table>

Restrictions

There is only one serious restriction to TLINK; TLINK does not generate Windows or OS/2 .EXE files.

Previous restrictions that no longer apply:

- 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 Turbo C++ (both the integrated environment and command-line versions), TASM, Turbo Prolog, and other compilers.

Error messages

TLINK has three types of errors: fatal errors, nonfatal errors, and warnings.

• A fatal error causes TLINK to stop immediately; the .EXE file is deleted.

• A nonfatal error does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file.

• 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.

<table>
<thead>
<tr>
<th>In manual</th>
<th>What you'll see on screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A file name (with or without extension)</td>
</tr>
<tr>
<td>group</td>
<td>A group name</td>
</tr>
<tr>
<td>module</td>
<td>A module name</td>
</tr>
<tr>
<td>segment</td>
<td>A segment name</td>
</tr>
<tr>
<td>symbol</td>
<td>A symbol name</td>
</tr>
<tr>
<td>XXXXh</td>
<td>A 4-digit hexadecimal number, followed by h</td>
</tr>
</tbody>
</table>

The error 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.

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.
Fatal errors
When fatal errors happen, TLINK stops and deletes the .EXE file.

`symbol in module module1 conflicts with module module2`
This error message can result from a conflict between two symbols (either public or communal). This usually means that a symbol with different attributes is defined in two modules. This is the same error message as the IDE message that reads: “`symbol conflicts with module module1 in module module2`”.

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.

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.

`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 (CO) object modules are being linked in.

`Cannot generate COM file : invalid initial entry point address`
You used the `/t` option, but the program starting address is not equal to 100H, which is required with .COM files.

`Cannot generate COM file : program exceeds 64K`
You used the `/t` option, but the total program size exceeds the .COM file limit.

`Cannot generate COM file : segment-relocatable items present`
You used the `/t` option, but the program contains segment-relative fixups, which are not allowed with .COM files.
Cannot generate COM file: stack segment present
You used the /t option, but the program declares a stack segment, which is not allowed with .COM files.

DOS error, ax = decimal 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.

Group group exceeds 64K
This message will occur if a group exceeds 64K bytes when the segments of the group are combined.

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.

Invalid group definition: group in module module
This error can occur if an attempt was made to assign a segment to more than one group. It can also result 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.

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.

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 Turbo C++, recompile the file. If the problem persists, contact Borland.

Not enough memory
There was not enough memory to complete the link process. Try removing any terminate-and-stay-resident applications currently loaded, or reduce the size of any RAM disk currently active. Then run TLINK again.

Relocation offset overflow in module module
This error only occurs for 32-bit object modules and indicates a relocation (segment fixup) offset greater than the DOS limit of 64K.
Relocation table full
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). (This error only occurs for 32 bit object modules.)

Segment segment exceeds 64K
This message will occur if too much data was defined for a given data or code segment, when segments of the same name in different source files are combined.

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 or external symbols.

32-bit record encountered in module module: use “/3” option
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.

Unable to open file filename
This occurs if the named file does not exist or is misspelled.

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.

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.

Nonfatal errors
TLINK has three nonfatal errors. As mentioned, when a nonfatal error occurs, the .EXE and .MAP files are not deleted. These errors are treated as fatal errors under the integrated environment.

Fixup overflow in module module, at segname:xxxxh, target = segment or symbol:xxhx
This indicates 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.

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 Turbo C++, there may be other possible causes for this message. Even in Turbo C++, this message could be generated if you are using different segment or group names than the default values for a given memory model. (In the IDE, this message reads: “Fixup overflow in segment segment in module module”.)

**Out of memory**
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.

**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 Turbo 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.

**Warnings**
TLINK has seven warning messages.

**Warning: symbol defined in module module1 is duplicated in module module2**
The named symbol is defined in each of the named modules. This could happen if a given object file is named twice in the command line.

**Warning: Group group1 overlaps group group2 in module module**
This means that TLINK has encountered nested groups. This warning only occurs when overlaps are used. (In the IDE, this message reads: “Segment segment is in two groups”.)
If a Turbo C++ program produces this message for any but the tiny memory model, make sure you are using the correct COX startup object files.

**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 Turbo C++, or for any application program that will be converted to a .COM file. For other programs, this indicates an error.

**Warning: No stub for fixup at segment:xxxxh in module**
This error occurs when the target for a fixup is in an overlay segment, but no stub segment is found for the segment. This is usually the result of not making public a symbol in an overlay that is referenced from the same module. (In the IDE, this message reads: “No stub for fixup at segment:xxxxxh in module module”.)

**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: Program entry point may not reside in an overlay**
This message usually results from specifying the \b option before the startup module (COX). COX contains the initial program entry point, which must not be overlaid. To fix this, simply specify the \b option following the startup module.

**Warning: Segment segment is in two groups: group1 and group2**
The linker found conflicting claims by the two named groups.

---

**TOUCH**

There are times when you want to force a particular target file to be recompiled or rebuilt, even though no changes have been made to its sources. One way to do this is to use the TOUCH utility. TOUCH changes the date and time of one or more files to the current date and time, making it “newer” than the files that depend on it.

You can force MAKE to rebuild a target file by touching one of the files that target depends on. To touch a file (or files), type

```
touch filename [filename ...]
```

You can use the DOS wildcards * and ? with TOUCH.
TOUCH

at the DOS prompt. TOUCH will then update the file's creation date(s). Once you do this, you can invoke MAKE to rebuild the touched target file(s).

**Important!** Before you use the TOUCH utility, it's vitally important to set your system's internal clock to the proper date and time. If you're using an IBM PC, XT, or compatible that doesn't have a battery-powered clock, don't forget to set the time and date using the DOS "time" and "date" commands. Failing to do this will keep both TOUCH and MAKE from working properly.
Customizing Turbo C++

Use TCINST to customize the integrated environment version of Turbo C++. Through TCINST, you can change various default settings in the integrated environment, such as the screen size, editing modes, menu colors, and default directories. In addition to letting you customize TC.EXE, TCINST also lets you specify and directly modify filename.EXE and .PRJ files. If you don’t specify a file, TCINST assumes TC.EXE. If TCINST doesn’t find the file you specified, it reports an error.

With TCINST, you can do any of the following:

- modify compiler option in a .PRJ file
- set up paths to the directories where your include, library, and output files are located
- choose default settings for the integrated debugger
- set defaults for the compiler and linker
- customize the editor command keys
- set up the editor defaults
- set up the default video display mode
- change screen colors

To make life easier for you, TCINST’s menus are quite similar to the menus in the integrated environment. Any option that you install with TCINST that also appears as a menu option in TC.EXE will be overridden whenever you load a configuration file that contains a different setting for that option, or when you change the setting via the menu system of the integrated environment. So
changes made to TC.EXE are only realized when no configuration files are loaded. For this reason, TCINST allows you to directly modify .PRJ files and the TCCONFIG.TC file.

Running TCINST

The syntax for TCINST is

```
TCINST [option] [exepath [exename] | [configpath]
TCCONFIG.TC | [prjpath]prjname.PRJ]
```

If you don't give a path and/or file name, TCINST looks for TC.EXE in the current directory. Otherwise, it uses the given path and file name.

`option` lets you specify whether you want to run TCINST in color (type `/c`) or in black and white (type `/b`). Normally, TCINST comes up in color if it detects a color adapter in a color mode. You can override this default if, for instance, you are using a composite monitor with a color adapter, by using the `/b` option.

**Note**

You can use TCINST to modify local copies of TCCONFIG.TC and .PRJ files. In this way, you can customize different copies of Turbo C++ on your system to use different editor command keys, different menu colors, and so on, by having different configuration files in your various project directories.

Using EGA with a CGA monitor

If you are running Turbo C++ on a system with an EGA display card and a CGA monitor, you must use TCINST to set Turbo C++ or it will not run properly. See page 210 for step-by-step instructions on how to do this.

The TCINST Installation menu

This chapter shows all possible customizable items. However, when you're installing a configuration file (.PRJ or .TC), only the menu items representing values in the configuration files are displayed. So when you're using TCINST to modify the TCCONFIG.TC file, compiler options are not available; when you're installing a .PRJ file, color customization is not available.

The first menu to appear on the screen is the TCINST Installation menu.
The **Search** option gives you access to the search defaults.

The **Run** option allows you to set default command-line arguments that will be passed to your running programs, exactly as if you had typed them on the DOS command line (redirection is not supported). It is only necessary to give the arguments here; the program name is omitted.

The **Options** command gives you access to default settings for a great many features, including memory model, degree of optimization, display of error messages, linker and environment settings, and path names to the directories holding header and library files.

You can use the **Editor Commands** option to customize the interactive editor's keystroke commands.

With **Mode for Display**, you can specify the video display mode that Turbo C++ will operate in, and whether yours is a "snowy" video adapter.

You can customize the colors of almost every part of the integrated environment through the **Adjust Colors** menu.

The **Save Configuration** option allows you a choice between saving and not saving changes to the TC.EXE file.

The **Quit** option asks if you want to quit without saving the changes you have made to the integrated development environment.

To choose a menu item, just press the key for the highlighted capital letter of the given option. For instance, press **A** to choose the **Adjust Colors** option. Or use the **↑** and **↓** keys to move the highlight bar to your choice, then press **Enter**.

Pressing **Esc** (more than once if necessary) returns you to the main installation menu.

---

**The Search menu**

- **Search**
  - **Direction**
  - **Scope**
  - **Origin**
  - **Case sensitive**
  - **Whole words only**
  - **Regular expression**
  - **Prompt on replace**

---

The **Search** menu has the following options:

- **Direction**: Set to either **Forward** or **Backward**.
- **Scope**: Set to either **Global** or **Selected text**.
- **Origin**: Set to either **From cursor** or **Entire scope**.
- **Case Sensitive**: Set **On** or **Off**
- **Whole Words Only**: Set **On** or **Off**
The Run menu

The Run menu option is Arguments. When you press Enter, a prompt box appears that allows you to set default command-line arguments that will be passed to your running programs, exactly as if you had typed them on the DOS command line (redirection is not supported). It is only necessary to give the arguments here; the program name is omitted.

The Options menu

In the Options menu you can set defaults for various features that determine how the integrated environment works.

The Full Menus menu

The Options | Full Menus command lets you use a subset of the complete set of menus in Turbo C++. Set Full menus Off to have the minimum command set appear in menus and dialog boxes.

Turbo C++ comes preset with Full menus Off, so that you initially see only the smaller set of menus. For more information on this command, see page 13.

The Compiler menu

The options in the Compiler menu allow you to set defaults for particular hardware configurations, memory models, code optimizations, diagnostic message control, and macro definitions.

Code Generation

The items in this menu let you set defaults for how the compiler will compile your source code. Each of these items can be set to On or Off, or to specific settings.

- Overlay Support
- Word Alignment
Duplicate Strings Merged
Unsigned Characters
Standard Stack Frame. If you plan to use the integrated debugger, turn this option On.
Test Stack Overflow
Model lets you choose the default memory model (method of memory addressing) that Turbo C++ will use. The options are Tiny, Small, Medium, Compact, Large, and Huge. Refer to Chapter 4 ("Memory models, floating point, and overlays") in the Programmer's Guide for more information about these memory models.
When you choose Defines, a prompt box appears in which you can enter macro definitions that will be available by default to Turbo C++.
The More options, each of which can be set to On or Off or to specific settings, are
- Generate Underbars
- Line Numbers Debug Info
- Debug Info in OBJs
- Assume SS not equal DS
- Treat enums as ints
- Fast Floating Point
- Floating Point: Set to either None, Emulation, 8087, or 80287.
- Instruction Set: Set to either 8088/8086, 80186, or 80286.
- Calling Convention: Set to either C or Pascal calling sequence.

C++ Options
The C++ Options are
- C++ Virtual Tables: Set to either Smart, Local, External, or Public.
- Use C++ Compiler: Set to either CPP extensions only or C++ always.
- Out-of-line Inline Functions: Set to either On or Off.
Optimizations

The options in the Optimizations menu let you set defaults for code optimization when your code is compiled.

- Register Optimization: Set On or Off.
- Jump Optimization: Set On or Off. If you plan to use the integrated debugger, turn this option Off.
- Register Variables: Set to either None, Register keyword, or Automatic.
- Optimize for: Set to Size or Speed.

Source

The Source menu options are

- Nested Comments: Set to On or Off.
- Keywords: Set to either Turbo C++, ANSI, UNIX V, or Kernighan and Ritchie.
- Identifier Length: Specify new identifier size (1 to 32).

Messages

The Message options are

- Errors: stop after: Specify new error size (0 to 255).
- Warnings: stop after: Specify new warning size (0 to 255).
- Display Warnings: Set to On or Off.
- The Portability options can each be set to On and Off. Your choices are
  - Non-Portable pointer conversion
  - Non-Portable pointer comparison
  - Constant out of range in comparison
  - Constant is long
  - Conversion may lose significant digits
  - Mixing pointers to signed and unsigned char
- The ANSI Violations options can each be set to On and Off. Your choices are
  - Void functions may not return a value
  - Both return and return of a value used
Suspicious pointer conversion
Undefined structure 'ident'
Redefinition of 'ident' is not identical
Hexadecimal value more than three digits
The More options can each be set to On and Off. Your choices are
Case bypasses initialization of a local variable
Goto bypasses initialization of a local variable
Untyped bit field assumed signed int
'ident' declared both external and static
Declare 'ident' prior to use in prototype
Division by zero
Initialization 'ident' with 'ident'
This initialization is only partially bracketed
Bit field must be signed or unsigned int
Declaration does not specify a tag or an identifier
The C++ Warnings can each be set to On and Off. Your choices are
Assignment to 'this' is obsolete
Base initialization without a class name
Functions containing 'ident' are not expanded inline
Function 'ident' should have a prototype
'ident' is both a structure tag and a name
Temporary used to initialize 'ident'
Temporary used for parameter 'ident'
The constant member 'ident' is not initialized
This style of function definition is now obsolete
Use of 'overload' is now unnecessary and obsolete
Obsolete syntax, use '::' instead
Assigning 'ident' to 'ident'
'ident' hides virtual function 'ident'
Non-const function 'ident' called for const object
The Frequent errors options can each be set to On and Off. Your choices are
Function should return a value
Important! Don't change this option unless you are an expert and have read Chapter 6 ("Interfacing with assembly language") in the Programmer's Guide.

- Unreachable code
- Code has no effect
- Possible use of 'ident' before definition
- 'Ident' is assigned a value which is never used
- Parameter 'ident' is never used
- Possibly incorrect assignment
- The More options can each be set to On and Off. Your choices are
  - Superfluous & with function or array
  - 'ident' declared but never used
  - Ambiguous operators need parentheses
  - Structure passed by value
  - No declaration for function 'ident'
  - Call to function with no prototype
  - Restarting compile using assembly
  - Unknown assembler instruction
  - Function definition cannot be a typedefed declaration
  - Ill formed pragma

Names

With the items in the Names menu, you can set the default segment, group, and class names for Code, Data, and BSS sections. When you choose one of these items, the asterisk (*) on the next menu that appears tells the compiler to use the default names.

The Make menu

- Break Make On lets you specify the default condition for stopping a make: if the file has Warnings, Errors, Fatal Errors, or Link.
- Check Auto-dependencies lets you set the default to On or Off.

The Linker menu

- Map File determines the default type for the map file. The choices are Off, Segments, Publics, or Detailed.
Set Initialize Segments On or Off. If this toggle is set to On, the linker initializes uninitialized segments.

Set Default Libraries On or Off. When you’re linking with modules that have been created by a compiler other than Turbo C++, the other compiler may have placed a list of default libraries in the object file. If this option is on, the linker tries to find any undefined routines in these libraries, as well as in the default libraries supplied by Turbo C++. If this option is set to Off, the linker will only search the default libraries supplied by Turbo C++; any defaults in .OBJ files will be ignored.

Graphics Library controls whether the linker links in BGI graphics library functions. This item defaults to On; set it Off to prevent the linker from searching GRAPHICS.LIB.

Warn Duplicate Symbols sets On or Off the linker warning for duplicate symbols in object and library files.

Stack Warning sets On or Off the “No stack specified” message generated by the linker.

Case-sensitive Link sets On or Off case sensitivity during linking. The usual setting is On, since C is a case-sensitive language.

Overlay EXE sets On or Off.

The Debugger menu

The items in the Debugger menu let you set certain default settings for the Turbo C++ integrated debugger.

When you compile your program with Source debugging set On, you can debug it using either the integrated debugger or with a standalone debugger, such as Borland’s Turbo Debugger. When it is set to Standalone, only the standalone debugger can be used. When it is set to None, no debugging information is placed in the .EXE file.

Display Swapping allows you to set the default level to None, Smart, or Always. When you run your program with the default setting Smart, the Debugger looks at the code being executed to see whether the code will affect the screen (that is, output to the screen). If the code outputs to the screen (or if it calls a function), the screen is swapped from the Editor screen to the Execution screen long enough for output to take place, then is swapped back. Otherwise, no swapping occurs. The Always setting causes the screen to be swapped every time a statement executes. The None setting causes the debugger not to swap the screen at all.
Program Heap Size specifies the new program heap size in K bytes from 4 through 640.

You have several choices for Inspector Options: Show Inherited (On or Off), Show Methods (On or Off), and/or Show Integers As. Show Integers As gives you the choice between Decimal, Hex, or Both.

The Directories menu

With Directories, you can specify a path to each of the integrated environment default directories. These are the directories Turbo C++ searches when looking for the include and library files, and the directory where it will place your program output.

When you choose Directories, TCINST brings up another menu. Select the item you wish to specify and enter the new path information into the input box. Enter the names for each of these just as you do for the corresponding menu items in the integrated environment. If you are not certain of each item’s syntax, refer to Chapter 1, “The IDE reference.”

After typing a path name (or names) for any of the directories menu items, press Enter to accept. When you exit the program, TCINST prompts you on whether you want to save the changes.

Include Directories

This option lets you specify default directories in which the Turbo C++ standard include (header) files and any project-specific header files are stored. A prompt box appears in which you can enter the directory names.

You can enter multiple directories in Include Directories. You must separate the directory path names with a semicolon (;), and you can enter a maximum of 127 characters with either menu item. You can enter absolute or relative path names; for example,

```
c:\turboc\lib; c:\turboc\mylibs; a\newturbo\mathlibs; a:\..\vidlibs
```

Library Directories

Use the Library Directories option to specify default directories for the Turbo C++ start-up object files (.OBJ) and run-time library files (.LIB) and any project-specific libraries. A prompt box appears in which you can enter the directory names.
As with Include Directories, you can enter multiple directories in Library Directories. You must separate the directory path names with a semicolon (; ), and you can enter a maximum of 127 characters with either menu item. You can enter absolute or relative path names.

Output Directory

Use this option to name the default directory where the compiler will store the .OBJ, .EXE, and .MAP files it creates.

The Output Directory menu item takes one directory path name; it accepts a maximum of 64 characters.

The Environment menu

With the items in the Environment menu, you can set defaults for various features of the Turbo C++ working environment.

Look at the Quick-Ref line for directions on how to choose these options. You can change the operating environment defaults to suit your preferences (and your monitor), then save them as part of Turbo C++. Of course, you'll still be able to change these settings from inside Turbo C++'s editor (or from the Options | Environment menu).

The Preferences menu

The Preference menu options are

The Screen Size menu allows you to specify whether your default integrated environment screen is to display 25 lines or 43/50 lines.

- **25 Lines.** This is the standard PC display: 25 lines by 80 columns. This is the only screen size available to systems with a Monochrome Display Adapter (MDA) or Color/Graphics Adapter (CGA).

- **43/50 Lines.** If your PC is equipped with an EGA or VGA, choose 43/50 lines to make your screen display 43 lines by 80 columns (for an EGA) or 50 lines by 80 columns (for a VGA).

Set Save Old Messages On or Off. This option determines whether error messages from earlier compiles are saved in the Message window or deleted.

Source Tracking determines whether a new window is opened or if an existing window is used when loading files during message
tracking and debugging. Set it to either New window or Current window.

Auto Save options are Editor files, Environment, Desktop, or Project. You can set them to On or Off.

The Editor menu

Use this menu to set defaults for various features of the integrated environment's editor.

- **Create Backup Files**: Toggle On or Off.
- **Insert Mode**: Toggle On or Off. With Insert mode On, the editor inserts anything you enter from the keyboard at the cursor position, and pushes existing text to the right of the cursor even further right. Toggling Insert mode Off allows you to overwrite text at the cursor.
- **Autoindent Mode**: Toggle On or Off. With Autoindent mode set to On, the cursor returns to the starting column of the previous line when you press Enter. When Autoindent mode is toggled Off, the cursor always returns to column one.
- **Use Tab Character**: Toggle On or Off. With Use tab character set to On, when you press the Tab key, the editor places a tab character (Ctrl-h) in the text, using the tab size specified with Tab Size. With Use tab character Off, when you press the Tab key, the editor inserts enough space characters to align the cursor with the first letter of each word in the previous line.
- **Optimal Fill**: Toggle On or Off. Optimal fill mode has no effect unless Use Tab Character is also set to On. When both these modes are enabled, the beginning of every autoindented and unindented line is filled optimally with tabs and spaces. This produces lines with a minimum number of characters.
- **Backspace Unindents**: Toggle On or Off. When it is set to On, this feature outdents the cursor; that is, it aligns the cursor with the first nonblank character in the first outdented line above the current or immediately preceding nonblank line.
- **Cursor Through Tabs**: Toggle On or Off.
- **Tab Size**: When you choose this option, a prompt box appears in which you can enter the number of spaces you want to tab over at each tab command. You can specify a Tab Size from 2 to 16.
The Mouse menu

Use this menu to set default for various mouse features.

- **Mouse Double Click**: Set to either Fast, Medium, or Slow.
- **Right Mouse Button**: Set to either Nothing, Topic search, Go To Cursor, Breakpoint, Inspect, Evaluate, or Add Watch.
- **Reverse Mouse**: Set On or Off.

The Editor Commands menu

Turbo C++'s interactive editor provides many editing functions, including commands for

- cursor movement
- text insertion and deletion
- block and file manipulation
- string search (plus search-and-replace)

These editing commands are assigned to certain keys (or key combinations); they are explained in detail in Chapter 3, “The editor from A to Z.”

When you choose Editor Commands from TCINST's main installation menu, the Install Editor screen comes up, displaying three columns of text.

- The first column (on the left) describes all the commands available in the interactive editor.
- The second column lists the Primary keystrokes: what keys or special key combinations you press to invoke a particular editor command.
- The third column lists the Secondary keystrokes: These are optional alternate keystrokes you can also press to invoke the same editor command.

The bottom lines of text in the Install Editor screen summarize the keys you use to choose entries in the Primary and Secondary columns.
### Table 6.1
Editor key commands

<table>
<thead>
<tr>
<th>Key</th>
<th>Legend</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>←, →, ↑ and ↓</td>
<td>Select</td>
<td>Selects the editor command you want to rekey.</td>
</tr>
<tr>
<td>Page</td>
<td>Page</td>
<td>Scrolls up or down one full screen page.</td>
</tr>
<tr>
<td>Enter</td>
<td>Modify</td>
<td>Enters the keystroke-modifying mode.</td>
</tr>
<tr>
<td>R</td>
<td>Restore</td>
<td>Resets all editor commands to the factory defaults</td>
</tr>
<tr>
<td>Esc</td>
<td>Exit</td>
<td>Leaves the Install Editor screen and returns to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the main TCINST installation menu.</td>
</tr>
<tr>
<td>F4</td>
<td>Key modes</td>
<td>Toggles between the three keystroke combinations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WordStar-like, Ignore case, and Verbatim.</td>
</tr>
</tbody>
</table>

After you press *Enter* to enter the modify mode, a pop-up window lists the current defined keystrokes for the chosen command, and the bottom lines of text in the Install Editor screen summarize the keys you use to change those keystrokes.

### Table 6.2
Customizing keystrokes

*To enter the keys F2, F3, or F4 as part of an editor command key sequence, first press the backquote key (‘), then the appropriate function key.*

<table>
<thead>
<tr>
<th>Key</th>
<th>Legend</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backspace</td>
<td>Backspace</td>
<td>Deletes keystroke to left of cursor.</td>
</tr>
<tr>
<td>Enter</td>
<td>Accept</td>
<td>Accepts newly defined keystrokes for the chosen editor command.</td>
</tr>
<tr>
<td>Esc</td>
<td>Abandon changes</td>
<td>Abandons changes to the current choice, restoring the command’s original keystrokes, and returns to the Install Editor screen (ready to choose another editor command).</td>
</tr>
<tr>
<td>F2</td>
<td>Restore</td>
<td>Abandons changes to current choice, restoring the command’s original keystrokes, but keeps the current command chosen for redefinition.</td>
</tr>
<tr>
<td>F3</td>
<td>Clear</td>
<td>Clears current choice’s keystroke definition, but keeps the current command chosen for redefinition.</td>
</tr>
</tbody>
</table>
Table 6.2: Customizing keystrokes (continued)

| F4 | Key modes | Toggles between the three keystroke combinations: WordStar-like, Ignore case, and Verbatim. |

There are three different keystroke combinations: WordStar-like, Ignore case, and Verbatim. These are listed on the bottom line of the screen; the highlighted one is the current choice. In all cases, the first character of the combination must be a special key or a control character. The active combination governs how the subsequent characters are handled.

- **WordStar-like**: In this mode, if you type a letter or one of the following characters:
  - [ ] \ ^ –
  it is automatically entered as a control-character combination. For example,
  - Pressing [ yields Ctrl-[  
  - Pressing a or A or Ctrl A yields Ctrl-A  
  - Pressing y or Y or Ctrl Y yields Ctrl-Y
  Thus, if you customize an editor command to be Ctrl-A Ctrl-B in WordStar-like mode, you can type any of the following in Turbo C++’s editor to activate that command:
    - Ctrl-A Ctrl-B  
    - Ctrl-A B  
    - Ctrl-A b

- **Ignore case**: In this mode, all alphabetic (letter) keys you enter are converted to their uppercase equivalents. When you type a letter in this mode, it is not automatically entered as a control-character combination; if a keystroke is to be a control-letter combination, you must hold down the Ctrl key while typing the letter. For example, in this mode, Ctrl-A B is not the same as Ctrl-A Ctrl-B.

- **Verbatim**: If you type a letter in this mode, it is entered exactly as you type it. So, for example, Ctrl-A Ctrl-B, Ctrl-A B, and Ctrl-A b are all distinct.

**Allowed keystrokes**

Although TCINST provides you with almost boundless flexibility in customizing the editor commands to your own tastes, there are a few rules governing the keystroke sequences you can define. Some of the rules apply to any
keystroke definition, while others come into effect only in certain keystroke modes.

1. You can enter a maximum of six keystrokes for any given editor command. Certain key combinations are equivalent to two keystrokes: These include Alt (any valid key); the cursor-movement keys (↑, ↓, PgDn, Del, and so on); and all function keys and their combinations (F4, Shift-F7, Alt-F8, and so on).

2. The first keystroke must be a character that is neither alphanumerical nor punctuation: that is, it must be a control key or a special key.

3. To enter the Esc key as a command keystroke, type Ctrl-\.

4. To enter the Backspace key as a command keystroke, type Ctrl-H.

5. To enter the Enter key as a command keystroke, type Ctrl-M.

6. The Turbo C++ predefined Help function keys (F1 and Alt-F1) can’t be reassigned as editor command keys. However, any other function key can. If you enter a Turbo C++ hot key as part of an editor command key sequence, TCINST issues a warning that you are overriding a hot key in the editor and verifies that you want to override that key.

Normally, Turbo C++ correctly detects your system’s video mode. You should only change the Mode for Display menu if one of the following holds true:

- You want to choose a mode other than the current video mode.
- You have a Color/Graphics Adapter that doesn’t “snow.”
- You think Turbo C++ is incorrectly detecting your hardware.
- You have a laptop or a system with a composite screen (which acts like a CGA with only one color). For this situation, choose Black and White.

If you choose Default, Turbo C++ always operates in the mode that is active when you load it.
If you choose **Color**, Turbo C++ uses 80-column color mode if a color adapter is detected, no matter what mode is active when you load TC.EXE, and switches back to the previously active mode when you exit.

If you choose **Black and White**, Turbo C++ uses 80-column black-and-white mode if a color adapter is detected, no matter what mode is active, and switches back to the previously active mode when you exit. Use this with laptops and composite monitors.

If you choose **LCD or Composite**, Turbo C++ uses 80-column black-and-white mode if a color adapter is detected, no matter what mode is active, and switches back to the previously active mode when you exit. Use this with laptops and composite monitors.

And if you choose **Monochrome**, Turbo C++ uses monochrome mode if a monochrome adapter is detected, no matter what mode is active.

When you choose one of the first four options, the program conducts a video test on your screen; refer to the Quick-Ref line for instructions on what to do. When you choose one of the options, the status line queries

> Conducting video test. Is your screen "snowy" now? Press any key to answer.

When you press any key, you can choose

- **Yes**, the screen was "snowy"
- **No**, always turn off snow checking
- **Maybe**, always check the hardware

Look at the Quick-Ref line for more about **Maybe**. Press Esc to return to the main installation menu.
The Adjust Colors menu

- Customize colors

- Main menu
- Pull-down menu
- Pop-up menus
- Status line
- Edit window
- Output window
- Message window
- Watch window
- Help
- Modal help
- Dialog box

- Main Menu: Choose between Normal text, First letter, Selection bar, or Selected letter.

- Pull-down Menu: Choose between Border, Normal text, First letter, Selection bar, Selected letter, Disabled entry, or Disabled selected.

- Pop-up Menus: Choose between Title, Border, Normal text, First letter, Selected bar, Selected letter, or Disabled entry.

- Status Line: Choose between Normal text or First letter.

- Edit Window: Choose between Active border, Inactive border, or Text.

- Output Window: Choose between Active border or Inactive border.

- Message Window: Choose between Active border, Inactive border, or Text.

- Watch Window: Choose between Active border, Inactive border, or Text.

- Help: Choose between Active border, Inactive border, or Text.

- Modal Help: Choose between Border or Text.

The Dialog Box options are

- Frame: Choose between Border, Title, or Background.

- Type ins: Choose between Destruct or Normal.

- Groups: Choose between Selected, Active border, Disabled, or First letters. From the First letters option, you can choose Selected or Active border.

- Messages: Choose between Information, Notification, Warning, or Errors.

- Buttons: Choose between Selected, Default, Active border, Disabled, or First letters. From the First letters options, you can choose Selected, Default, or Active border.

- Lists: Choose between Active border, Default, Selected, Disabled, Scroll bars, Arrow head, or Thumb tabs.
Check boxes: Choose between Selected, Active border, Disabled, or First letters. From the First letters options, you can choose Selected or Active border.

Labels: Choose between Selected, Active border, Disabled, Normal text, or First letters. From the First letters options, you can choose Selected or Active border.

Once you have made all desired changes, choose Save Configuration at the main installation menu. The message

Save changes to TC.EXE? (Y/N):

appears at the bottom of the screen. When modifying TCCONFIG.TC or .PRJ files, no prompting is done.

If you press Y (for Yes), all the changes you have made are permanently installed into the file you’re modifying. (You can always run TCINST again if you want to change them.)

If you press N (for No), your changes are ignored and you are returned to the main menu without Turbo C++’s defaults or startup appearance being changed.

You can restore the Editor Commands by choosing the E option at the TCINST main menu, then press R (for Restore factory defaults) and Esc.

Quit menu

When you choose Quit at the main installation menu, the message

Quit without saving? (Y/N):

appears at the bottom of the screen.

If you press Y (for Yes), your changes are ignored and you are returned to the operating system prompt.

If you press N (for No), you are returned to the Installation menu.
Turbo Editor macros

The Turbo Editor Macro Language (TEML) is a powerful utility that you can use to enhance or change the Turbo C++ editor. Using the 140-odd built-in macros, you can define new ones that perform sophisticated editing tasks and that can bind keystrokes to these tasks.

Operation

In order to use TEML, you first write a macro script in a text editor. You then compile the script using the Turbo Editor Macro Compiler (TEMC). The compiled file is used as a configuration file in Turbo C++.

The Turbo Editor Macro Compiler expects as input an ASCII file containing definitions and binding conforming to the TEML specification. The output is placed in a configuration file that can be used by the IDE. The changes from TEMC are incremental; this means that if you just change the definition of one key, only that key will be changed in the configuration file. Everything else will stay as it was.

Here is the syntax for the TEMC utility:

    TEMC scriptfile outputconfigfile

You can use any text editor (including Turbo C++'s) to create the ASCII scriptfile. You use the outputconfigfile by naming it
TCCONFIG.TC and placing it in the directory you will be in when starting TC.EXE.

Editor macro language syntax

TEML has a simple syntax based on Pascal and C. Here are the basic syntax rules of the macro language:

- Statements in a script file are separated with a semicolon.
- Reserved words in TEML are:
  - ALT
  - CTRL
  - MACRO
  - SHIFT
  - BEGIN
  - END
  - SCRIPT
- Comments are designated in the C style between /* and */ marks.
- In strings, the user can place any legal C backslash (\) sequence; for example, "\xD".

The rest of this section describes how each possible component of the syntax fits into the overall scheme. In this list, the symbol ::= means that the object on the left side is composed of the objects on the right side. If the list of objects on the right side of the ::= begins with the vertical bar symbol, then the object on the left can be composed of nothing or one of the listed items.

Script ::= ScriptName ScriptItems

ScriptName ::= | SCRIPTIdentifier ;

ScriptItems ::= | ScriptItems ScriptItem

ScriptItem ::= KeyAssignment | MacroDefinition

KeyAssignment ::= KeySequence : Command ;

KeySequence ::= KeySpecifier | KeySequence + KeySpecifier | KeySequence + ^ KeySpecifier

KeySpecifier ::= Key | KeyModifier Key

Key ::= Number | Identifier | END

KeyModifier ::= | CTRL | ALT | SHIFT -
Command ::= BEGIN CommandList OptSemicolon END | MacroCommand

CommandList ::= Command | CommandList ; Command

MacroCommand ::= CommandName | CommandName (ParamList)

CommandName ::= Identifier

ParamList ::= Param | ParamList , Param

Param ::= Number | String

MacroDefinition ::= MACRO CommandName CommandList OptSemicolon END ;

OptSemicolon ::= | ;

Number ::= Digit | Number Digit

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Identifier ::= Letter | Identifier LetterDigit

Letter ::= A to Z | a to z | _

LetterDigit ::= Letter | Digit

String ::= " AnyCharacterNotQuote "

Example scripts

This example sets up a host of WordStar-like keyboard shortcuts.

Script WordStar;

Macro NewLine
   RightOfLine;
   InsertText("\xD");
End;

/* Key Assignments */
Ctrl-A : WordLeft;
Ctrl-C : PageDown;
Ctrl-D : CursorCharRight;
Ctrl-E : CursorUp;
Ctrl-F : WordRight;
Ctrl-G : DeleteChar;
Ctrl-H : BackSpaceDelete;
Ctrl-J : CursorDown;
Ctrl-K+B : SetBlockBeg;
Ctrl-K+C : CopyBlock;
Ctrl-K+H : ToggleHideBlock;
Ctrl-K+K : SetBlockEnd;
Ctrl-K+Q : Exit;
Ctrl-K+R : ReadBlock;
Ctrl-K+V : MoveBlock;
Ctrl-K+W : WriteBlock;
Ctrl-K+Y : DeleteBlock;
Ctrl-K+1 : SetMark(1);
Ctrl-K+2 : SetMark(2);
Ctrl-K+3 : SetMark(3);
Ctrl-L : RepeatSearch;
Ctrl-N : BreakLine;
Ctrl-O : NewLine; /* This is not a WordStar keystroke */
Ctrl-P : LiteralChar;
Ctrl-Q+A : Replace;
Ctrl-Q+B : MoveToBlockBeg;
Ctrl-Q+C : EndCursor;
Ctrl-Q+D : RightOfLine;
Ctrl-Q+E : TopOfScreen;
Ctrl-Q+F : GetFindString;
Ctrl-Q+K : MoveToBlockEnd;
Ctrl-Q+P : MoveToPrevPos;
Ctrl-Q+R : HomeCursor;
Ctrl-Q+S : LeftOfLine;
Ctrl-Q+X : BottomOfScreen;
Ctrl-Q+Y : DeleteToEol;
Ctrl-Q+1 : begin
    MoveToMark(1);
    CenterFixScreenPos;
end;

Ctrl-Q+2 : begin
    MoveToMark(2);
    CenterFixScreenPos;
end;

Ctrl-Q+3 : begin
    MoveToMark(3);
    CenterFixScreenPos;
end;

Ctrl-R : PageUp;
Ctrl-S : CursorCharLeft;
Ctrl-T : DeleteWord;
Ctrl-V : ToggleInsert;
Ctrl-W : ScrollDown;
Ctrl-X : CursorDown;
MakeFuncText

MakeFuncText creates a commented area for descriptive text associated with a function, assumes the cursor is positioned immediately after the name, and the name is at the left of the screen.

Script util;

macro MakeFuncText
    InsertText("\n\n"); /* add some whitespace */
    CursorUp;
    CursorUp;
    LeftOfLine; /* go before beginning of intended function name */
    LeftOfLine;
    SetBlockBeg;
    WordRight;
    SetBlockEnd;
    LeftOfLine;
    CursorDown;
    CopyBlockRaw; /* copy for prototyping */
    CursorUp;
    LeftOfLine;
    InsertText("\nFunction "); /* add "Function" to comment area */
    RightOfLine;
    InsertText(":"); /* .. and colon at end */
    CursorUp; /* put in comment lines fore and aft */
    LeftOfLine;
    InsertText("/**********");
    InsertText("**********");
    CursorDown;
    RightOfLine;
    InsertText("\n");
    InsertText("\tDescription:\n");

Appendix A, Turbo Editor macros 219
MakeStub

MakeStub creates a stub, based on a user-entered function name; it assumes the cursor is positioned immediately after the name, and the name is at the left of the screen.

macro MakeStub
LeftOfLine;          /* go before beginning of intended */
function name */
InsertText("void ");       /* put in void return type and param */
RightOfLine;
InsertText("( void )\n\n");
InsertText("\t");
InsertText("printf("This is ");       /* start printf statement */
CursorUp;             /* go back to function name */
CursorUp;
LeftOfLine;
WordRight;
SetBlockBeg;          /* mark function name */
WordRight;
CursorLeft;
CursorLeft;
SetBlockEnd;
CursorDown;           /* go back to printf statement */
CursorDown;
RightofLine;
InsertText(" ");
CopyBlockRaw;         /* put it in printf statement */
SetBlockBeg;
SetBlockEnd;          /* clear marked block */
RightofLine;
InsertText("\n\n");
InsertText("\n");      /* add newline and closing brace */
end;                 /* MakeStub */

Alt-S         : MakeStub;
/* This one doesn't conflict with default assignments */
Built-in commands

The names of the built-in commands describe their actions. Commands with the word *screen* in them generally only affect the screen.

Commands that have the word *raw* in them perform fewer housekeeping tasks than their “raw-less” counterparts. For example, in a long macro, using raw commands saves time in that they don’t constantly update the screen display to reflect each change in cursor position. However, you would only use the raw macros as intermediate steps in combination with other macros.

Macro names are not case-sensitive. A few macros require parameters in parentheses, as discussed in the descriptions.

Remember, you can use these primitive macros to build more complicated ones.

Functional index

This section lists the built-in macros by function. The following section is a straight alphabetical list.

### Block macros

These macros affect blocks of text.

*You should use SetPrevPos or FixScreenPos, or both, at the end of the raw macros for housekeeping purposes.*

- CopyBlock
- DeleteBlock
- DeleteBlockRaw
- HighlightBlock
- MoveBlock
- MoveToBlockBeg
- MoveToBlockEnd
- MoveToBlockBegRaw
- MoveToBlockEndRaw

### Deletion/insertion

These macros delete, undelete, and insert text.

- BackspaceDelete
- ClipClear
- ClipCopy
- ClipCut
- ClipPaste
- ClipShow
- DeleteBlock
- DeleteBlockRaw
- DeleteChar
- DeleteLine
- DeleteToEOL
- DeleteWord
- EditMenu
- InsertText
- LiteralChar
- RestoreLine
SetInsertMode  ToggleInsert  

**Search macros**  These macros deal with searching.

- GetFindString
- MatchPairForward
- MatchPairBackward
- RepeatSearch
- Replace
- SearchMenu

**Hot key macros**  These macros duplicate the hot keys in the IDE.

- AddWatch
- CloseWindow
- CompileFile
- Help
- HelpLine
- Inspect
- LastHelp
- MakeProject
- Menu
- Modify
- NextError
- NextWindow
- OpenFile
- PrevError
- ResetProgram
- RunProgram
- RunToHere
- SaveFile
- SetBreakpoint
- Step
- Trace
- ViewCallStack
- ViewUserScreen
- ZoomWindow

**Menu macros**  These macros pull down the various menus in the IDE.

- CompileMenu
- DebugMenu
- EditMenu
- FileMenu
- HelpMenu
- OptionsMenu
- RunMenu
- SearchMenu
- SystemMenu
- WindowsMenu

**Screen movement**  These macros control cursor movement and screen movement.

- BottomOfScreen
- BottomOfScreenRaw
- CenterFixScreenPos
- CursorCharLeft
- CursorCharRight
- CursorDown
- CursorLeft
- CursorRight
- CursorUp
- EndCursor
- EndCursorRaw
- FixCursorPos
- FixScreenPos
- HomeCursor
- HomeCursorRaw
- LeftOfLine
- MoveToMark
- MoveToPrevPos
- PageDown
- PageUp
- PageScreenDown
- PageScreenUp
RightOfLine
ScrollDown
ScrollUp
ScrollScreenDown
ScrollScreenUp
SetMark

SetPrevPos
SwapPrevPos
TopOfScreen
TopOfScreenRaw
WordLeft
WordRight

System macros
These macros affect certain system functions.
Exit
FullPaintScreen
PaintScreen
Quit
SmartRefreshScreen

Window macros
These macros affect windows in the IDE.
CloseWindow
GotoWindow1
GotoWindow2
GotoWindow3
GotoWindow4
GotoWindow5
GotoWindow6
GotoWindow7
GotoWindow8
GotoWindow9
SwapWindows
WindowList
WindowsMenu
ZoomWindow

Alphabetical reference
This section is an alphabetical list of all the built-in macros. If you need to see how the macros are grouped by function, refer to the preceding section.

AddWatch
This macro is the same as pressing Ctrl-F7 or Debug | Watches | Add Watch.

BackspaceDelete
Moves the cursor back one character and deletes it (typically defined to be Backspace).

BottomOfScreen
Moves the cursor position to the lower left corner of the screen. This macro automatically sets the starting cursor position so that you can go back there with the MoveToPrevPos macro.

BottomOfScreenRaw
Moves the cursor to the lower left corner of the screen. As opposed to the BottomOfScreen macro, this command does not change the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CenterFixScreenPos</td>
<td>Corrects the screen image position relative to the cursor. This command moves the screen image so that the cursor is in the middle of it.</td>
</tr>
<tr>
<td>ClipClear</td>
<td>Removes the selected text but does not change the Clipboard. This macro is the same as pressing Ctrl-Del or choosing Edit</td>
</tr>
<tr>
<td>ClipCopy</td>
<td>Copies the selected text so you can paste a copy of it elsewhere. This macro is the same as pressing Ctrl-Ins or choosing Edit</td>
</tr>
<tr>
<td>ClipCut</td>
<td>Cuts the selected text. This macro is the same as pressing Shift-Del or choosing Edit</td>
</tr>
<tr>
<td>ClipPaste</td>
<td>Pastes the last-cut or last-copied text. This macro is the same as pressing Shift-Ins or choosing Edit</td>
</tr>
<tr>
<td>ClipShow</td>
<td>Opens the Clipboard window.</td>
</tr>
<tr>
<td>CloseWindow</td>
<td>Closes the current window. This macro is the same as pressing Alt-F2 or choosing the Window</td>
</tr>
<tr>
<td>CompileFile</td>
<td>Compiles the current file. This macro is the same as pressing Alt-F9 or choosing the Compile</td>
</tr>
<tr>
<td>CompileMenu</td>
<td>Pulls down the Compile menu.</td>
</tr>
<tr>
<td>CopyBlock</td>
<td>Inserts a copy of the current block at the cursor position. Unlike the CopyBlockRaw macro, this macro highlights the new block.</td>
</tr>
<tr>
<td>CursorCharLeft</td>
<td>Moves the cursor one character to the left. (If the cursor is at the beginning of a line, this command makes it wrap to the previous printing character.)</td>
</tr>
<tr>
<td>CursorCharRight</td>
<td>Moves the cursor one character to the right. (If the cursor is at the end of a line, this command makes it wrap to the next printing character.)</td>
</tr>
<tr>
<td>CursorDown</td>
<td>Moves the cursor one line down, keeping it in the same column.</td>
</tr>
<tr>
<td>CursorLeft</td>
<td>Moves the cursor one column to the left.</td>
</tr>
<tr>
<td>CursorRight</td>
<td>Moves the cursor one column to the right (even if there are no characters there). If the cursor is at the edge of the screen, this command moves the cursor off the visible screen.</td>
</tr>
<tr>
<td>CursorUp</td>
<td>Moves the cursor one line up, keeping it in the same column.</td>
</tr>
<tr>
<td>DebugMenu</td>
<td>Pulls down the Debug menu.</td>
</tr>
</tbody>
</table>
DelelBlock**

Deletes the current block. Unlike the DeleteBlockRaw macro, DeleteBlock leaves the cursor fixed in one spot on the screen (it doesn’t move when the block is deleted).

DeleteBlockRaw**

Deletes the current block. Unlike the DeleteBlock macro, this “raw” macro doesn’t fix the cursor in one spot on the screen (it can move when the block is deleted).

DeleteChar**

Deletes the character at the cursor position.

DeleteLine**

Deletes the line the cursor is on.

DeleteToEOL**

Deletes from the cursor position to the end of the line.

DeleteWord**

Deletes the word the cursor is on plus the space characters after it.

EditMenu**

Pulls down the Edit menu.

EndCursor**

Moves the cursor to the end of the file. This macro automatically sets the previous cursor position so that you can go back there with the MoveToPrevPos macro.

EndCursorRaw**

Moves the cursor to the end of the file. As opposed to the EndCursor macro, this command does not reset the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

Exit**

Exits from the editor.

FileMenu**

Pulls down the File menu.

FixCursorPos**

Corrects the cursor position in respect to the screen. This command moves the cursor to the visible screen by making the least amount of movement possible, the result being that the cursor appears at the start or the end of the screen.

FixScreenPos**

Corrects the screen position in respect to the cursor. This command moves the screen image to the cursor by making the least amount of movement possible, the result being that the screen appears above or below the cursor position.

FullPaintScreen**

Forces a full refresh of the screen. This paints out to the edge of the screen; it is slower than PaintScreen.

GetFindString**

Opens the Find dialog box so you can search for a text string. The search begins at the current cursor position.

GotoWindow1, GotoWindow2, ..., GotoWindow9**

Makes the specified window active (the window number is in the upper right corner). These macros are the same as pressing Alt-1, Alt-2, and so on.
Help
Opens the Help window, just like the Help | Table of Contents command. This macro is the same as pressing F1.

HelpMenu
Pulls down the Help menu.

HighlightBlock
Highlights the current marked block.

HomeCursor
Moves the cursor position to the beginning of the file. This macro automatically sets the starting cursor position so that you can go back there with the MoveToPrevPos macro.

HomeCursorRaw
Moves the cursor to the beginning of the file. As opposed to the HomeCursor macro, this command does not change the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

InsertText("string")
Inserts string at the current cursor position. The double quotes are required around string; string can be up to 4,096 characters long.

Inspect
This macro is the same as pressing Alt-F4 or Debug | Inspect.

LastHelp
Opens the Help window that was last viewed, just like the Help | Previous Topic command. This macro is the same as pressing Alt-F1.

LeftOfLine
Moves the cursor to the beginning of the line (typically defined to be Home).

LiteralChar
Inserts the next key pressed verbatim into the file (such as Ctrl-P).

MakeProject
This macro is the same as pressing F9.

MatchPairBackward
Finds the matching delimiter character that complements the one at the current cursor position. Searches backward (to the beginning) in the file. See page 110 for a complete list of delimiters.

MatchPairForward
Finds the matching delimiter character that complements the one at the current cursor position. Searches forward (to the end) in the file. See page 110 for a complete list of delimiters.

Menu
Makes the menu bar active. This macro is the same as pressing F10.

Modify
This macro is the same as pressing Ctrl-F4 or Debug | Evaluate/Modify.

MoveBlock
Moves the current block to the cursor position. Unlike the MoveBlockRaw macro, this macro highlights the new block.
**MoveToBlockBeg** Moves the cursor to the beginning of the current block. Unlike the MoveToBlockBegRaw macro, this macro updates the cursor on the screen and changes the “previous cursor" location, which you access with the SwapPrevPos and MoveToPrevPos macros.

**MoveToBlockBegRaw** Moves the cursor to the beginning of the current block. Unlike the MoveToBlockBeg macro, this “raw” macro doesn’t update the cursor onscreen and doesn’t change the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

**MoveToBlockEnd** Moves the cursor to the end of the current block. Unlike the MoveToBlockEndRaw macro, this macro updates the cursor onscreen and changes the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

**MoveToBlockEndRaw** Moves the cursor to the end of the current block. Unlike the MoveToBlockEnd macro, this “raw” macro doesn’t update the cursor onscreen and doesn’t change the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

**MoveToMark(number)** Moves the cursor to the location designated by the SetMark(number) macro. You can set 10 marks by passing SetMark a parameter of 0 to 9. You move the cursor to any of the 10 marks by passing the corresponding number (0-9) to the MoveToMark(number) macro.

**MoveToPrevPos** Moves the cursor to the position designated by the SetPrevPos macro.

**NextError** Moves to the next error position. This macro is the same as pressing Alt-F8 or choosing the Search | Next Error command.

**NextWindow** Makes the next window active, just like the Window | Next command. This macro is the same as pressing F6.

**OpenFile** Displays the Open dialog box. This macro is the same as pressing F3.

**OptionsMenu** Pulls down the Options menu.

**PageDown** Scrolls both the screen and cursor down one page.

**PageScreenDown** Moves the screen down one screenful, possibly moving the cursor out of view (typically defined to be PgDn).

**PageScreenUp** Moves the screen up one screenful, possibly moving the cursor out of view (typically defined to be PgUp).
PageUp  Scrolls both the screen and cursor up one page. (Typically defined to bePgUp.)

PaintScreen  Forces a full refresh of the screen. PaintScreen only paints lines from the buffer; it assumes it knows how to blank end-of-lines. It’s faster than FullPaintScreen.

PrevError  Moves to the previous error position. This macro is the same as pressing Alt-F7 or choosing the Search | Previous Error command.

Quit  Exits from the integrated environment. If you’ve made changes you haven’t saved, you’ll be given a chance to save them before quitting. This macro is the same as pressing Alt-X.

ReadBlock  Lets you open a text file and insert it at the cursor position. The ReadBlock macro automatically opens the Open dialog box so you can choose a file to open.

RepeatSearch  Searches for the text string that was last entered in the find dialog box using the GetFindString macro.

Replace  Opens the Replace dialog box so you can search for and replace text.

ResetProgram  Reset the current program. This macro is the same as pressing Ctrl-F2 or choosing Run | Program Reset.

RestoreLine  Inserts the line deleted with the DeleteLine macro. If the cursor has moved to another line since the DeleteLine macro, this macro does nothing.

RightOfLine  Moves the cursor to the end of the line (typically defined to be End).

RunMenu  Pulls down the Run menu.

RunProgram  Runs the current program. This macro is the same as pressing Ctrl-F9 or choosing Run | Run command.

RunToHere  Runs a program up to the line containing the cursor. This macro is the same as pressing F4 or choosing Run | Go to Cursor.

SaveFile  Saves the file in the current window. This macro is the same as pressing F2 or choosing the File | Save command.

ScrollDown  Scrolls the screen down one line. This macro will not allow the cursor to scroll out of view.
ScrollScreenDown Moves the screen down one line, leaving the cursor at the same relative position in the file. This command will allow the cursor to scroll out of view.

ScrollScreenUp Moves the screen up one line, leaving the cursor at the same relative position in the file. This command will allow the cursor to scroll out of view.

ScrollUp Scrolls the screen up one line. This command will not allow the cursor to scroll out of view.

SearchMenu Pulls down the Search menu.

SetBlockBeg Marks the current cursor position as the beginning of a block. Unlike the SetBlockBegRaw macro, this macro highlights the new block.

SetBlockEnd Marks the current cursor position as the end of a block. Unlike the SetBlockEndRaw macro, this macro highlights the new block.

SetBreakpoint Sets a breakpoint at the cursor position. This macro is the same as pressing Ctrl-F8 or choosing Debug | Toggle Breakpoint.

SetInsertMode Turns insert mode on. To turn it off, type

BEGIN SetInsertMode; Toggle Insert END;

SetMark(number) Sets the current cursor position so that you can return to it using the MoveToMark(number) macro. You can set number to any number from 0 to 9. You move the cursor to any of the 10 marks by passing the corresponding number (0-9) to the MoveToMark(number) macro.

SetPrevPos Marks the current cursor position as the place to return to when you use the SwapPrevPos or MoveToPrevPos macros. Many macros implicitly set the "previous position" (the notable exceptions are "raw" macros).

SmartRefreshScreen Refreshes only the parts of the screen that have changed.

Step Runs a program one statement at a time but stepping over subroutines. This macro is the same as pressing F8 or choosing Run | Step Over.

SwapPrevPos Switches the current cursor position with the spot designated by the SetPrevPos macro.

SystemMenu Pulls down the System menu.

ToggleHideBlock Highlights or hides the current marked block.
**ToggleInsert**  Switches insert modes, from Insert to Overwrite or from Overwrite to Insert.

**TopOfScreen**  Moves the cursor to the upper left corner of the screen. This macro automatically sets the previous cursor position so that you can go back to it with the MoveToPrevPos macro.

**TopOfScreenRaw**  Moves the cursor to the upper left corner of the screen. As opposed to the TopOfScreen macro, this command does not change the “previous cursor” location, which you access with the SwapPrevPos and MoveToPrevPos macros.

**Trace**  Runs a program one statement at a time, moving into subroutines as necessary. This macro is the same as pressing F7 or choosing Run | Trace Into.

**ViewCallStack**  This macro is the same as pressing Ctrl-F3 or Debug | Call Stack.

**ViewFullOutput**  Switches views to the User Screen. This macro is the same as pressing Alt-F5 or choosing Window | User Screen command.

**WindowList**  Displays a list of all open windows. This macro is the same as pressing Alt-0.

**WindowsMenu**  Pulls down the Windows menu.

**WordLeft**  Moves the cursor one word to the left, placing it on the first character of that word.

**WordRight**  Moves the cursor one word to the right, placing it on the first character of that word.

**WriteBlock**  Lets you save the current block to a file. The WriteBlock macro automatically opens the Write Block to File dialog box so you can enter a file name.

**ZoomWindow**  Resizes the current window to be as large as possible, or—if the window is already zoomed—to be its original size. This macro is the same as pressing F5.

---

**Error messages**

While coding your macros, you may encounter certain errors. Knowing the compiler capacity may help you avoid some of those errors, which are given after this list of memory requirements.

- each macro invocation takes 1 byte
- each integer parameter takes 2 bytes
- each character parameter takes $(\text{number\_of\_characters\_in\_string} + 1)$ byte
- each macro requires 1 byte for \texttt{end}

\textbf{Cannot allocate memory for file.}
Not enough memory is available to process the file. TEMC needs about 100K of available space to compile a file.

\textbf{Expected item.}
The line indicated is most likely missing the specified item.

\textbf{File filename could not be created.}
The file specified for output cannot be created. Either the disk is full or you do not have rights to the current network drive or the name specified is not legal.

\textbf{File filename is empty.}
The file passed to TEMC to compile has nothing in it.

\textbf{File filename larger than 64K.}
The script file is larger than the maximum 64K in size.

\textbf{File filename not found.}
The file specified does not exist.

\textbf{Invalid key.}
Key specified is not valid.

\textbf{Invalid symbol symbol.}
The symbol specified is not a valid TEMC symbol.

\textbf{Out of memory.}
Not enough memory is available to process the file. TEMC needs about 100K of available space to compile a file.

\textbf{Read error on file filename.}
TEMC could not read the file source file.

\textbf{Redefinition of key.}
This key is defined elsewhere in the file.

\textbf{Redefinition of macro macro.}
This macro is defined elsewhere in the file.

\textbf{Parameters to a macro call illegal.}
Macros cannot have parameters. Trying to pass a parameter to a macro is, therefore, illegal.

\textbf{Script too complex.}
One or more of the following conditions need to be corrected:
- Too many keys defined
- String parameter is too long (the maximum string length is 256 characters)
- Too many parameters
- Macro size may be too large (the maximum size allowed is 1,024 bytes)

**Undefined symbol** `symbol`.
The symbol specified has not yet been defined.

**Unexpected item**.
The indicated line most likely would be correct if the item specified was deleted or changed.

**Unexpected end of file**.
The last macro or BEGIN/END pair was not terminated.

**Warning message**

**Redefinition of environment hot key**.
The key being defined is a hot key in the environment.
Redefining a hot key in the script will change its meaning in the editor only.
### 43/50-line display 75, 205
### 8088 processor
- instruction set 199
- + and + (TLIB action symbols) 173
-<> (angle brackets) in #include directive 74
-$* (base file name macro) 158
-» (chevron) in dialog boxes 19
-. (file name and extension macro) 159
-$& (file name only macro) 159
-$< (full file name macro) 158
-? MAKE help option 141
-1 option (extended 80186 instructions) 119,
  See also 80186 processor, generating extended instructions
-3 TLINK option (32-bit code) 186
-* and *- (TLIB action symbols) 173
-32-bit code 186
-(hyphen) MAKE command (ignore exit status) 146

### 25-line display 75, 205
# (MAKE comment character) 145
&& operator
  MAKE 146, 147
<< operator
  MAKE 146
>> operator
  MAKE 146
; (semicolons) in directory path names 75
= (System) menu 8
(~ (tilde) in transfer program names 65
* (TLIB action symbol) 173
+(TLIB action symbol) 173
-(TLIB action symbol) 173
$ editor macros See individual names of macros; transfer macros
@ MAKE command 146
80x87 math coprocessors See numeric coprocessors

### 80x86 processor
- 32-bit code 186
  extended instructions 119
  instruction set 55, 199
≡ (System) menu 8, 21
→ (arrows) in dialog boxes 18

### A
- a MAKE option (autodependency check) 141, 152
- a TCC option (align integers) 120
- A TCC option (ANSI keywords) 124
About command 21
ACBP field 183
action symbols See TLIB (librarian)
activating
  menu bar 9
active window See windows, active
add (TLIB action symbol) 173
Add button 50
Add Item command 50, 89
Add Item to Project List dialog box 50
Add Watch command 46
  hot key 13
  macro 223
AddWatch editor macro 223
Adjust Colors menu, TCINST 197, 212
Advanced Code Generation dialog box 55
alignment
  attribute 183
  integers 120
  word 53
  TCINST and 198
allowed keystrokes 209
American National Standards Institute See ANSI
angle brackets (<> in #include directive 74
ANSI

compatible code 124
floating point conversion rules 120
keywords 200
option 124
using only 59
Turbo C++ keywords and 124
violations 125
ANSI Violations 61
ANSI Violations options, TCINST 200
applications
transferring to and from Turbo C++ 64
Arguments
command 37
dialog box 37
arguments
command-line compiler 113
variable list 121
Arguments option, TCINST 198
arrays, inspecting values 41
arrows (+) in dialog boxes 18
.ASM files See assembly language
assembly language
assembling 113
compiling 128
default assembler 128
directory 130
inline routines 128
assembling 113
output files 128
projects and 94
assembly level debugger See Turbo Debugger
Assume SS not equal DS option 56
Assume SS not equal DS option, TCINST 199
attributes
ACBP 183
alignment 183
big 183
combining 183
-AU option (UNIX keywords) 124
Auto Save option 76
Auto Save option, TCINST 206
auto variables See variables, automatic
.AUTODEPEND MAKE directive 160
autoindent mode
changing default 206
Autoindent Mode option 77

Autoindent Mode option, TCINST 206
automatic dependencies 70
checking 94
MAKE (program manager) 141, 152
TCINST option 202
MAKE option 160
automatic variables See variables, automatic

B
/b IDE option (build) 6
-B MAKE option (build all) 141
-b options (allocate whole word for enums) 56
-b TCC option (allocate whole word for enums) 120
-B TCC option (process inline assembler code) 128
backspace macro 223
Backspace Unindents option 77
Backspace Unindents option, TCINST 206
BackSpaceDelete editor macro 223
backup files (.BAK) 77
backward
pair matching 109
searching 31
.BAK files 77
bar, title 15
base file name macro (MAKE) 158
batch files, MAKE 147, 148
BBS segment See also segments
class 202
group 202
renaming 202
BGI See Borland Graphics Interface
big attribute 183
Black and White option, TCINST 211
block operations (editor) See editing, block operations
Borland Graphics Interface (BGI) See also graphics
EGA palettes and 8
library 71
BottomOfScreen editor macro 223
BottomOfScreenRaw editor macro 223
boxes See check boxes; dialog boxes; list boxes; text, boxes
Break Make On
menu 92
option 70
Break Make On menu, TCINST 202
Breakpoints
command 47
dialog box 47
breakpoints See also debugging
clearing 48
controlling 47
deleting 47
editing 47
inline functions and 57
losing 48
setting 47
macro 229
viewing 47
BSSnames 63
bugs See debugging
Build All command 38
build IDE option 6
BUILTINS.MAK 140
buttons
Change 108
Change All 32, 108
choosing 18
in dialog boxes 18
mouse 78
radio 19

C
C++ 57, See also C language; Turbo C++
compiling 57
functions
inline
command-line option (−vi) 122
debugging and 57, 122
help 83
virtual tables 57, 199
warnings 62, 126, 201
C++ Code Generation
command 56
C++ Options menu, TCINST 199
C++ Virtual Tables option, TCINST 199
C++ Warnings
dialog box 62
C++ Warnings options, TCINST 201
−c TCC option (compile but don’t link) 128
−C TCC option (nested comments) 124
/C TLIB option (case sensitivity) 171, 175
/c TLINK option (case sensitivity) 184
C language See also C++
calling sequence 199
help 83
Turbo C++ and 121
C0x.OBJ 180
C0x.OBJ start-up object file 130
call stack
viewing
macro 230
Call Stack command 44
hot key 13
calling
conventions 55
sequences 199
Calling Convention option, TCINST 199
Cancel button 18
$CAP EDIT macro 65
$CAP EDIT transfer macro 68
$CAP MSG transfer macro 68
Cascade command 80
Case-sensitive Link option, TCINST 203
Case Sensitive option, TCINST 197
case sensitivity
in searches 30
linking with 72
TLIB option 171, 175
TLINK and 184
cdecl statement 121
CenterFixScreenPos editor macro 224
.CFG files See configuration files
Change All button 32, 108
Change button 108
Change Dir command 25
Change Directory dialog box 25
char treated as type unsigned 121
characters
control
IDE and 19
data type char See data types, char
literal
macro 226
tab
printing 25
Check Auto-dependencies option, TCINST 202
Check Autodependencies 70
check boxes 19
chevron symbol (») 19
Class Inspector window 42
classes
debugging 42
names 63
Clear command 29, 105
hot key 12
macro 224
Clear Desktop command 21
click speed (mouse) 78
Clipboard 28
editing text in 29
opening
macro 224
showing 29
ClipClear editor macro 224
ClipCopy editor macro 224
ClipCut editor macro 224
ClipPaste editor macro 224
ClipShow editor macro 224
close boxes 15
Close command 80
hot key 12
macro 224
Close Project
command 50
CloseWindow editor macro 224
Code Generation
command 53
dialog box 53, 55
code generation
command-line compiler options 118, 119
Code Generation menu, TCINST 198
code segment
class 202
group 128, 202
names 63
naming and renaming 127, 202
default 128
$COL transfer macro 67
Color/Graphics Adapter (CGA)
EGA card and 196
snow and 210
Color option, TCINST 210
columns
numbers 14, 67

.COM files
generating 178, 180
TLINK 185
limitations 185
size 185
combining attribute 183
command line
arguments 113
TCINST and 198
options See command-line compiler, options;
integrated environment, command-line options
Turbo C++ See command-line compiler
viewing from IDE 80, 81
command-line compiler 113-136
command format 113
compiling and linking with 113
configuration files See configuration files, TCC
MAKE and 160
options 114, 117
-1 (extended 80186 instructions) 119
-a (align integers) 120
-AK (Kernighan and Ritchie keywords) 124
allocate whole word for enum (-b) 56, 120
ANSI
compatible code 124
keywords (-A) 124
violations 125
assembler code 128
assembler to use (-E) 128
-AU (UNIX keywords) 124
-b (allocate whole word for enums) 56, 120
-B (process inline assembler) 128
C++ inline functions (-vi) 122
-c (compile and assemble) 128
-C (nested comments) 124
code generation 118, 119
code segment
class name 127
group 128
name 128
comments, nesting (-C) 124
compilation control 118, 128
compile and assemble (-c) 128
configuration files and 116
-D (macro definitions) 119
-d (merge literal strings) 120
data segment
  class name 128
  group 127, 128
  name 127, 128
desktop information (-v) 57, 122, 184
#define 119
ganging 119
-E (assembler to use) 128
-e (EXE program name) 129
emulate 80x87 (-f) 120
environment 130
error reporting 118, 125
.EXE file names 129
expanded and extended memory (-Q) 129
extended 80186 instructions (-1) 119
-f87 (inline 80x87 code) 120
-f (emulate 80x87) 120
fast floating point (-ff) 56, 120
frequent errors 126
functions, void 125
-G (speed optimization) 123
generate underscores (-u) 121
gn (stop on n errors) 125
identifiers, length (-i) 124
include files 131
  directory (-1) 130, 132
inline 80x87 code (-f87) 120
integer alignment (-a) 120
-JN stop on n messages) 125
-k (standard stack frame) 121
-K (unsigned characters) 121
Kernighan and Ritchie keywords (-AK) 124
-l (linker options) 130
-L (object code and library directory) 130, 132, 134
libraries 131
  directory (-L) 130, 132
line numbers (-y) 122
link map (-M) 130
linker (-l) 129, 130
-M (link map) 130
macro definitions (-D) 118, 119
memory model (-mx) 117, 118
merge literal strings (-d) 120
-n (.OBJ and .ASM directory) 130
-N (stack overflow logic) 121
nested comments (-C) 124
object code and library directory (-L) 130, 132, 134
object files (-o) 128
optimization (-O) 118, 123
order of evaluation 134
Pascal
  conventions (-p) 121
  identifiers 121
pointer conversion, suspicious 125
portability warnings 126
precedence 134
process inline assembler (-B) 128
produce .ASM but don't assemble (-S) 128
project files and 51
-Q TCC option (expanded and extended memory) 129
-r (register variables) 123
-rd (register variables) 123
redundant load operations 123
register variables 123
-S (produce .ASM but don't assemble) 128
segment-naming control 118, 127
source code 118
speed optimization (-G) 123
stack overflow error message (-N) 121
standard stack frame (-k) 121
stop on n errors (-gn) 125
stop on n messages (-jn) 125
structures and 125
symbolic debugger 122
syntax 116
toggling 116
Turbo C++ keywords (-A-) 124
underscores (-u) 121
UNIX keywords (-AU) 124
undefine (-U) 119
-v (debugging information) 57, 122, 184
-vi (C++ inline functions) 122
warnings (-wxxx) 125-127
-y (line numbers) 122
-Z (aliasing) 123
-zX (code and data segments) 127
-zX (code and data segments) 127, 128
response files 117
TLINK and 181
Turbo Assembler and 117
commands See also command-line compiler, options; individual command names
choosing
  with a mouse 9
  with keyboard 9
editor 102-108
  block operations 103, 104-106
cursor movement 102, 104
insert and delete 103
printing
  MAKE option 160
comments
delimiters 109
  pair matching 110, 111
in makefiles 145
nested 59, 124
  TCINST option 200
comparisons
pointers
  non-portable 200
compilation 118
command-line compiler options 118, 128
rules governing 116
Compile
command
  macro 224
menu 37
  macro 224
Compile to OBJ command 38
  hot key 13
CompileFile editor macro 224
CompileMenu editor macro 224
Compiler
  command 52
compiler
  Turbo editor macro 215
Compiler menu, TCINST 198
Compiler Messages dialog box 60
compilers See also compiling
C++ 57
  code optimization 58
command line See command-line compiler
configuration files See configuration files
customizing 198
memory models See memory models
optimizations
  for speed or size 58
stopping after errors and warnings 60
Turbo editor macro 67
compiling See also compilers
current file
  macro 224
to .EXE file 38
to .OBJ file 38
composite screens
customizing Turbo C++ for 210
conditional breakpoints See breakpoints, conditional
conditional execution directives (MAKE) 144, 162
  expressions in 163
$CONFIG transfer macro 67
configuration files 67, See also TCONF.TC
current, saved automatically 206
IDE 84-86
  modifying 196
projects and 84
  TCONF.TC 85
saving 79
TCC 132, 133
  converting 135
creating 134
  multiple 135
overriding 116, 134
TCINST overridden by 195
constants
debugging 42
hexadecimal
  TCINST option 201
  too large 125
long 200
  manifest See macros
octal
  too large 125
out of range 200
symbolic See macros
Contents command 83
  hot key 12
control characters
entering in IDE 19
format specifier 45
conventions, typographic 2
conversions
  floating point
    ANSI rules 120
  pointer
    suspicious 200
  pointers
    non-portable 200
    suspicious 125
  significant digits and 200
  specifications See format specifiers
coprocessors See numeric coprocessors
Copy command 28, 105
  hot key 12
  macro 224
Copy Example command 28, 83
CopyBlock editor macro 224
copying, and pasting See editing, copy and paste
copyright information 21
.CPP files See C++
CPU registers 81
CPx.LIB 130
Create Backup Files option 77
Create Backup Files option, TCINST 206
creating new files See files, new
Ctrl-Break 33, 34
Current window option 76
cursor See also editor, cursor movement
  running programs to
    macro 228
Cursor Through Tabs option 77
Cursor Through Tabs option, TCINST 206
CursorCharLeft editor macro 224
CursorCharRight editor macro 224
CursorDown editor macro 224
CursorLeft editor macro 224
CursorRight editor macro 224
CursorUp editor macro 224
Customize colors menu, TCINST 212
customizing See also TCINST
  code generation 198
  compiler 198
  editing commands 207
  EGA 196
  IDE 75
  keystroke commands 197
  multiple versions of Turbo C++ 196
  optimization 200
  order of precedence of commands 196
  quitting 213
Cut command 28, 105
  hot key 12
  macro 224
Cx.LIB 130, 180

D
/d IDE option (dual monitors) 7
$D MAKE macro (defined test) 157
  expressions and 164
-D MAKE option (define identifier) 141, 155
-D TCC option (macro definitions) 119
-d TCC option (merge literal strings) 120
/d TLINK option (duplicate symbols) 184
data
  aligning 53
data members See C++, data members
data segment
  class 202
  group 127, 128, 202
  names 63
  naming and renaming 127, 128
  renaming 202
data structures See also arrays; structures
data types
  char
    default 53
  characters
    unsigned
      TCINST and 199
  converting See conversion
  floating point See floating point
  integers See integers
Debug Info in OBJs option 56
  Trace into command and 36
Debug Info in OBJs option, TCINST 199
Debug menu 39
  macro 224
debugger, integrated See integrated debugger
Debugger command 72
Debugger menu, TCINST 203
Debugger Options dialog box 72
debugging See also integrated debugger
  arrays 41
breakpoints  See breakpoints
call stack 44
classes 42
constants 42
Debug Info in OBJs 56
dialog box choices 72
display swapping 73
  dual monitors and 73
excluding information 51
expressions 43
format specifiers 44
functions 42
heap size 74
hot keys 13
information 34, 72, 184
  command-line compiler option 122
  in .EXE or OBJ files 122
  in .EXE or .OBJ files 199
  storing 56
inspecting values 39
line numbers information 56
MAKE 141
map files 182
pointers 40
screen swapping 203
stack overflow 54
starting a session 33
Step Over command 36
  macro 229
structures and unions 41
subroutines 53
TLINK and 186
Trace Into command 35
  macro 230
types 43
variables 43
watchpoints
  adding 46
    macro 223
  controlling 45
  deleting 46, 47
  editing 46
  watch window 81
DebugMenu editor macro 224
default assembler 128
default buttons 18
Default Extension option 77
Default Extension option, TCINST 206
Default Libraries option 71
Default Libraries option, TCINST 203
Default option, TCINST 210
#define directive
  command-line compiler options 118, 119
  ganging 119
defined test macro (MAKE) 157
Defines option 54
Defines option, TCINST 199
Delete Item
  command 50
Delete Item command 89
Delete Watch command 46
DeleteBlock editor macro 225
DeleteBlockRaw editor macro 225
DeleteChar editor macro 225
DeleteLine editor macro 225
DeleteToEOL editor macro 225
DeleteWord editor macro 225
deleting line
  undoing 28
delimiters
  directional 109
  nesting 110
  nondirectional 109
  unmatched 111
dependencies 70
  checking
    MAKE (program manager) 152
desktop
  clearing 21
desktop files (.DSK)
    default 86
    projects and 86
Desktop option 76
Dialog Box options
  TCINST 212
dialog boxes
  Add Item to Project List 50
Advanced Code Generation 55
ANSI Violations 61
Arguments 37
arrows in 18
Breakpoints 47
buttons  See buttons
C++ Warnings 62
Change Directory 25
check boxes  See check boxes
Code Generation 53
Compiler Messages 60
Debugger Options 72
defined 18
Directories 74
entering text 19
Environment Options 103, 106
Find 30, 107
Frequent Errors 63
Get Info 26
Go to Line Number 33
Include Files 52
Linker 71
list boxes  See list boxes
Load a File 22, 106
Locate Function 33
More ANSI Violations 62
More Frequent Errors 63
Optimizations Options 58
Override Options 51
Portability 61
Preferences 75
Project File 49
radio buttons  See radio buttons
Replace 32, 108
Save File As 24
Segment Names 64
Source Options 59, 111
Transfer 64
$DIR transfer macro 68
Direction option, TCINST 197
directional delimiters  See delimiters
directional pair matching 109
directives 144
  MAKE  See MAKE (program manager),
directives
Directories
  command 74
dialog box 74
directories
  changing 25
defining 74
header files 204
include files 130, 132, 204
  example 132
MAKE 141
libraries 131, 204
  command-line option 130, 132
  example 132
output 75, 205
pick file 204
project files 85
projects 91
semicolons in paths 75
transfer macro 68
Directories menu, TCINST 204
disk caches
display
  expanded and extended memory and 129
display
  formats
  debugger 44
  repaint 22
  swapping 73
dual monitors and 73
Display Swapping option, TCINST 203
Display Warnings option 60
Display Warnings option, TCINST 200
DOS
  commands
    MAKE and 149
eenvironment strings
    macros and 157
MODE command 7
output
  viewing from IDE 80, 81
paths
  MAKE 160
  redirection 198
wildcards 23
DOS Shell command 8, 26
double-click speed (mouse) 78
$DRIVE transfer macro 68
DS register (data segment pointer) 56
.DSK files
  default 86
  projects and 86
dual monitor mode 7
dual monitors 7
display swapping and 73
DOS command line and 27
duplicate, strings, merging 53
Duplicate Strings Merged option 53
Duplicate Strings Merged option, TCINST 198
duplicate symbols 72
.LIB and .OBJ files and 185
TLINK and 184
warning
toggle 203

E
/e IDE option (extended memory) 7
-E TCC option (assembler to use) 128
-e TCC option (EXE program name) 129
/E TLIB option (extended dictionary) 171, 174
/e TLINK option 185
Edit
command (Turbo C 2.0) 101
menu 27
macro 225
windows
loading files into 93
Edit Watch command 46
Edit window
TCINST option 212
Edit windows
activating 101
cursor
moving 102, 104
option settings 76
editing 20, See also editor; text
autoindent mode 106
setting default 206
block operations 103, 104-106
deleting 105
deleting text
macro 225
hiding/unhiding 105
macro 226, 229
macros 221-230
marking
macro 229
moving
macros 226
printing 105
reading and writing 105
reading file in
macro 228
save (write) block to file
macro 230
selecting blocks 27, 105
breakpoints 47
clear command
macro 224
Clipboard text 29
commands 102-108
cursor movement 102, 104
customizing 207
insert and delete 103
copy and paste 105, See also Clipboard
hot key 12
macro 224
create backup files
setting default 206
cut and paste 28, 105
macros 222, 224
deleting text
macro 225
entering text 102
extension
setting default 206
fill
setting default 206
hot keys 12, 102-108
insert mode
macro 229
overwrite mode vs. 77
setting default 206
insertion and deletion
macros 221
macros 215-230
error messages 230
memory requirements 230
modifying 226
matching pairs See pair matching
miscellaneous commands 106
overwrite mode
macro 229
pair matching See pair matching
pasting See editing, copy and paste
place marker 106
print file 105
quitting 106
macro 228
restore line 106
macro 228
search and replace 107-108
    macros 222, 225, 228
    options 107
selecting text 27
strings
    inserting 226
tab mode toggle 106
tabs 106
    setting default 206
undelete 106
    macro 228
undoing line edits 28
unindent
    setting default 206
watchpoints 46
EditMenu editor macro 225
editor See also editing
    allowed keystrokes 209
cursor movement macros
    beginning or end of block 227
    beginning or end of file 225, 226
    beginning or end of line 226, 228
    character left or right 224
    column left or right 224
    line up or down 224
    to marker 227
    previous position 229
    word left or right 230
cursor position macros
    beginning or end of file 226
    marking 229
    screen 223, 224, 225
    setting 229
customizing 207
defaults 206
features 20
inspecting
    macro 226
macro language (TEML) 215-230
    alphabetical list 223-230
built-in macros 221
    blocks of text 221
    editing and searching 222
    functional list 221
    hot keys 222
    insertion and deletion 221
    menus 222
    screen movement 222
    system functions 223
    window management 223
combining macros 221
error messages 230
example scripts 217
memory requirements 230
modifying 222
syntax 216
using 67, 215
macros See also MAKE (program manager),
    macros
quitting
    macro 225, 228
    setting defaults 206
tabs in 77
    TCINST and 197
Editor Commands option, TCINST 207
Editor Files option 76
Editor menu, TCINST 206
Editor Options 76
SEDNAME transfer macro 67
EGA See Enhanced Graphics Adapter
!elif MAKE directive 162
    defined test macro and 157
    macros and 157
ellipsis mark (...) 9, 18
!else MAKE directive 162
EMS memory 26
EMU.LIB 130, 180, 181
emulation, 80x87 120
floating point 55
EndCursor editor macro 225
EndCursorRaw editor macro 225
!endif MAKE directive 162
Enhanced Graphics Adapter (EGA) 76
    CGA monitor and 196
    screen size 205
Enhanced Graphics Adapters (EGA)
    palette
    IDE option 8
enumerations
    assigning integers to 125
    treating as integers 199
enumerations (enum)
    command-line option 120
Environment
command 75
environment, DOS  See also integrated environment
macros and 157
Environment menu, TCINST 205
Environment option
Auto Save 76
Environment Options dialog box 103, 106
$ERRCOL transfer macro 67
$ERRLINE transfer macro 67
$ERRNAME transfer macro 67
!error MAKE directive 165
errors  See also warnings
ANSI 125
Frequent 62
frequent 126
MAKE (list) 166-169
messages
compile time 91, 92
keeping 205
removing 93
saving 93
searching 33
setting 60
next
hot key 13, 93
macro 227
previous
hot key 13, 93
macro 228
reporting
command-line compiler options 118, 125
stopping on n 60
setting n 200
syntax
project files 91, 92
tracking 205
TLINK (list) 188
tracking
project files 91, 92
Turbo Editor Macro Language 230
Errors : Stop After option, TCINST 200
Esc shortcut 18
Evaluate command
format specifiers and 44
Evaluate/Modify command 43
hot key 13
evaluation order
command-line compiler options 134
examples
copying from Help 28, 83
library and include directories 132
MAKE (program manager) 141
batch files 148
Turbo Editor macros 217
.EXE files
creating 38
debugging information 186
directory 75
file name transfer macro 67
linking 38
naming 38
overlaying 203
user-selected name for 129
executable files  See .EXE files
$SEXENAME transfer macro 67
exit codes
MAKE and 146
Exit editor macro 225
exiting Turbo C++ 8
expanded and extended memory
controlling use of 129
disk caches and RAM disks and 129
TLINK and 187
explicit
library files 131
rules (MAKE) 144, 150
expressions
debugging 43
evaluating
restrictions on 43
MAKE and 163, 164
nested
pair matching 108
values
displaying 43
$EXT transfer macro 68
extended 80186 instructions 119
extended and expanded memory 7
RAM disk and 7
extended dictionary
TLIB and 171, 174
extended memory  See expanded and extended memory
extension keywords
   ANSI and 124
extensions, file, supplied by TLINK 177
External option
   C++ Virtual tables 57
extract and remove (TLIB action) 173

F
   -f87 option (inline 80x87 code) 120
   -f MAKE option (MAKE file name) 139, 141
   -f TCC option (emulate 80x87) 120
Fast Floating Point option 56
Fast Floating Point option, TCINST 199
fatal errors  See errors
features
   editor 20
      integrated environment 5
   -ff option (fast floating point) 56, 120
file-inclusion directive (!include) 161
File menu 22
   macro 225
file-name macros (MAKE) 159
FileMenu editor macro 225
files  See also individual file-name extensions
   assembly language  See assembly language
backup (.BAK) 77
batch  See batch files
C++  See C++
closed
   reopening 81
.COM 178, 180
compiling
   macro 224
configuration  See configuration files
.CPP  See C++
desktop (.DSK)
   default 86
      projects and 86
   editing  See editing
executable  See .EXE files
extensions 68, 177
include  See include files
information in dependency checks 94
information on 26
library  See libraries, files
loading into editor 93
make  See MAKE (program manager)
map  See map files
multiple  See projects
names
   extensions  See projects
      meanings 177
   macros 68
      transfer 66
new 24, 106
NONAME 24
open
   choosing from List window 81
opening 22, 106
   hot key 11
      macro 227
out of date, recompiled 94
path
   macros 68
   printing 25
project (.PRJ)  See projects
response  See response files
saving 24, 106
   all 24, 69
      automatically 76
   hot key 11
   macro 228
      with new name or path 24
source
   .ASM 113
   .TC  See configuration files, IDE
   updating 138
filling lines with tabs and spaces 77
filters 68
   GREP and TASM 68
Find command 29, 107,  See also searching
   hot key 12
Find dialog box 30, 107
   macro 225
FixCursorPos editor macro 225
FixScreenPos editor macro 225
floating point  See also integers; numbers;
   numeric coprocessors
   ANSI conversion rules 120
code generation 55
   emulation 199
fast 56, 120
TCINST option 199
format specifier 45
inline 80x87 operations 120
libraries 120
TLINK and 181
math coprocessor and 121
Floating Point option, TCINST 199
format specifiers
documenting and 44
table 45
43/50-line display 75, 205
forward
forward searching 31
pair matching 109
FP87.LIB 130, 180, 181
Frequent Errors
warnings 62
frequent errors 62, 126
Frequent Errors dialog box 63
Frequent Errors options, TCINST 201
full file name macro (MAKE) 158
full link map 130
Full Menus command 13, 52
Full Menus menu, TCINST 198
FullPaintScreen editor macro 225
functions  See also scope
  C-type 121
call stack and 44
calling conventions 55
help 83
inline
  out of line 199
inspecting 42
locating 33
parameters  See arguments
searching for 33
stepping over 36
tracing into 35
void
  returning a value 125
TCINST option 200
ganging
  command-line compiler options
  #define 119
  macro definition 119
defined 119, 131
  integrated environment 131
  library and include files 131
Generate Underbars option 56
Generate Underbars option, TCINST 199
Get Info
  command 26
dialog box 26
GetFindString editor macro 225
global menus  See menus
global variables
  word-aligning 120
Go Cursor command 35
Go To Cursor command
  macro 228
Go to Cursor command
  hot key 11, 13
Go to Line Number
  command 32
dialog box 33
GoToWindowN editor macro 225
graphics
  library
    TLINK and 180
  palette
    EGA 8
Graphics Library option 71
Graphics Library option, TCINST 203
GREP (file searcher)
capturing messages 68
  wildcards in Turbo C++ 30
GREP2MSG.EXE 68
group names 63

H
  -h MAKE option (help) 141
header files  See also include files
directory 204
help 83
  searching for 131
heap
  size 74
Help
button 18
text editor macro 226
menu 82
   macro 226
TCINST option 212
windows
closing 82
copying from 28, 83
keywords in 82
macro 226
opening 82
selecting text in 82
help
accessing 82
C and C++ 83
hot keys 11, 12
index 83
keywords 82
language 83
MAKE 141
previous topic 84
macro 226
status line 17
table of contents 83
Help on Help command 84
HelpMenu editor macro 226
hexadecimal numbers See numbers, hexadecimal
HighlightBlock editor macro 226
history lists 19
closing 21
wildcards and 23
HomeCursor editor macro 226
HomeCursorRaw editor macro 226
hot keys
debugging 13
ing 12
text editor 101, 102-108
help 11, 12
macros 222
make project 92
menus 10, 11
next error 93
previous error 93
redefining 210
transfer macros 65
transfer program names 65
using 10
hyphen (-- MAKE command (ignore exit status) 146

-I MAKE option (ignore exit status) 141
-1 MAKE option (include files directory) 140, 141
-i TCC option (identifier length) 124
-I TCC option (include files directory) 130, 132, 134
/i TLINK option (uninitialized trailing segments) 184
icons
full menus 13
IDE See integrated environment
Identifier Length option, TCINST 200
identifiers
defining 155
duplicate 72
warning 203
length 59
setting 200
Pascal-type 121
significant length of 119, 124
Turbo C++ keywords as 59, 124
undefined 119
underscore for 121
!if MAKE directive 162
defined test macro and 157
macros and 157
Ignore Case option, TCINST 209
ignore exit status (MAKE command) 146
!IGNORE MAKE directive 160
implicit
library files 131
rule (MAKE) 144
#include directive See also include files
angled brackets and 131
directories 74
quotes and 131
#include directive (MAKE) 140, 161
Include Directories
input box 74
Include Directories option, TCINST 204
Include Files
   command 51
include files See also header files
   automatic dependency checking (MAKE) 152
command-line compiler options 131
directories 130, 132, 204
   multiple 132
help 83
MAKE 140, 161
directories 141
projects 89
searching for 131
user-specified 130, 132
Include Files command 89
Include Files dialog box 52
incremental search 20
indenting automatically 77
Index command
   hot key 12
Index command (help) 83
indexes See arrays
initialization modules
   used with TLINK 180
initialization modules, used with TLINK 179
Initialize Segments option, TCINST 202
initialized data segment See data segment
inline assembly code 128
inline code See assembly language, inline
   routines; 80x87 math coprocessor
input boxes 19
Insert Mode option 77
Insert Mode option, TCINST 206
InsertText editor macro 226
Inspect
   command 39
   editor macro 226
Inspect command
   hot key 12, 13
Inspector Options option, TCINST 204
Inspector windows 39
   arrays 41
class 42
constant 42
function 42
ordinal 40
pointers 40
structures and unions 41
Type 43
Install Editor screen 207, 208
Installation menu, TCINST 196
Instruction Set, TCINST 199
Instruction Set radio buttons 55
integers See also floating point; numbers
   aligned on word boundary 120
   assigning to enumeration 125
integrated debugger 199, 203, See also
debugging
   breakpoints See breakpoints
debugging information for 122
integrated environment 5
command-line arguments and 37
command-line options 6
   build (/b) 6
dual monitors (/d) 7
EGA palette (/p) 8
expanded memory (/x) 7
extended memory (/e) 7
laptops (/l) 8
make (/m) 6
/p (EGA palette) 8
RAM disk (/n) 7
syntax 6
/x expanded memory) 7
configuration files See configuration files,
   IDE
customizing 75, 195, 205
debugging See debugging
   editing See editing
   features 5
ganging 131
default 93
memory needs 70
menus See menus
monitor
make files 93
multiple library directories 131
invoking
TCINST 196

J
   -j TCC option (stop on n messages) 125
Jump Optimization
   option 58
Jump Optimization option, TCINST 200

K
-K MAKE option (keep temporary files) 141, 147
-k TCC option (standard stack frame) 121
-K TCC option (unsigned characters) 121
K&R See Kernighan and Ritchie
Keep Messages command
toggle 93
Kernighan and Ritchie
keywords 59, 124, 200
keyboard
choosing buttons with 18
choosing commands with 9
selecting text with 27
keys, hot See hot keys
keystrokes
allowed (in customizing) 209
commands
customizing 197, 207, 208
Ignore case 209
types of 209
Verbatim 209
WordStar-like 209
keywords
ANSI 200
command 124
help 83
Help windows 82
Kernighan and Ritchie 200
using 124
options 59
register
Register Variables option and 58
Turbo C++ 59, 200
using, as identifiers 124
UNIX
using 124
UNIX V 200
Keywords option, TCINST 200

L
/l IDE option (LCD screen) 8
-l TCC option (linker options) 130
-L TCC option (object code and library directory) 130, 132, 134
/l TLINK option (line numbers) 184
language help 83
laptop computers
customizing Turbo C++ for 210
laptops
integrated environment option (/l) 8
LastHelp editor macro 226
LCD or Composite option, TCINST 211
left-handed
mouse support for 78
LeftOfLine editor macro 226
.LIB files See libraries
libname (TLIB option) 171
librarian See TLIB
libraries
command-line compiler options 131
creating 171
default 71, 203
directories 75, 130, 204
command-line option 130, 132
multiple 132
duplicate symbols in 185
explicit and implicit 131
files 75, 130, 132
user-specified 130
floating point 120
TLINK and 181
graphics 71, 203
TLINK and 180
linking 38
math
project library overrides and 97
memory models 181
memory models and 180-181
numeric coprocessor 181
object files 169, 170
creating 173
order of use 180
overriding in projects 97
page size 174
rebuilding 122
routines
80x87 floating-point emulation 120
run time
TLINK and 181
searching for 131, 132
TLINK and 179, 180
ignoring 184
user-specified 131
utility See TLIB
Library Directories
input box 75
Library Directories option, TCINST 204
library files See libraries
$LINE transfer macro 67
line numbers See lines, numbering
Line Numbers Debug Info option 56
Line Numbers Debug Info option, TCINST 199
lines
filling with tabs and spaces 77
moving cursor to 32
numbering 14, 67, 199
in object files 122
information for debugging 56
TLINK and 184
restoring (in editor) 28
Link EXE File command 38
link map, full 130
Linker
command 71
dialog box 71
linker See also TLINK
case sensitive linking 72
command-line compiler options 129, 130
link map
creating 130
linking from command line 113
options
from command-line compiler 130
Linker menu, TCINST 202
linking
excluding from 51
list boxes 20
file names 23
searching incrementally 83
List command
hot key 12
List window 79, 81
listfile (TLIB option) 171
literal strings See strings, literal
LiteralChar editor macro 226
Load a File dialog box 22, 106
load operations
redundant, suppressing 123
local menus See menus
Local option
C++ Virtual tables 57
Local Options
command 50
Local Options command 89
Locate Function
command 33
dialog box 33
.LST files See files; listfile (TLIB option)
M
/m IDE option (make) 6
-M TCC option (link map) 130
/m TLINK option (publics in map file) 178
macros See also editor, macros; MAKE
(program manager), macros
command-line compiler 118, 119
definitions
default 199
DOS
environment strings and 157
path (MAKE) 160
editor 67
ganging 119
invocation
defined 156
MAKE See MAKE (program manager),
macros
preprocessor 54
transfer See transfer macros
Turbo editor See editor, macro language
(TEML)
Main Menu option, TCINST 212
__MAKE__ macro 157
MAKE (Program Manager)
IDE makes and 93
stopping makes 92
MAKE (program manager) 138-169
automatic dependency checking 141, 152
batching files and 147
BUILTINS.MAK file 140
clocks and 139
commands
@ (hide commands) 146
Index
stopping makes 70
swapping in memory 160
syntax 139
wildcards and 150
Make command 70
Make EXE command
  hot key 11, 13
Make EXE File command 38
makefiles See MAKE (program manager)
MakeProject editor macro 226
makes
  IDE option 6
manifest constants See macros
Map File
  TCINST 202
map files 130
  debugging 182
  directory 75
  generated by TLINK 178, 182
  options 71
Match Pair command 108-112
MatchPairBackward editor macro 226
MatchPairForward editor macro 226
math coprocessors See numeric coprocessors
MATHx.LIB 130, 180
maximize See Zoom command
$MEM transfer macro 68, 69
memory
  addressing 199
  available 26
  dump
    format specifier 45
EMS 26
expanded and extended 129
  controlling 129
  disk caches and RAM disk and 129
  TLINK option 187
extended and expanded 7
  RAM disk and 7
heap size 74
requirements
  IDE 70
swapping MAKE in 160
transfer programs 68, 69
Turbo Editor Macro Language requirements 230
memory models
  changing 54
  command-line options 56, 117, 118
  default
    changing 199
    initialization modules 180
  libraries 181, 180-181
  startup modules 181
tiny
    library 181
  TLINK and 179, 180
Menu
  editor macro 226
menu bar See menus
menus See also individual menu names
  accessing 9
  bar
    macro 226
  Break Make On 92
  commands See individual command names
  editor macros for 222
Full 13, 52
  how indicated in manual 13
  TCINST and 195
hot keys 10, 11
items
  choosing 197
  opening 9, 101
  Optimization 123
  TCINST and 195
  with arrows (↑) 9
  with ellipsis marks (...) 9, 18
Message Tracking
  toggle 92
Message window 80, 93
  capturing output into 68
  removing messages 38
  TCINST option 212
messages See also errors; warnings
  appending 76
  capturing from programs 68
  column number 67
  file name 67
  line number 67
  removing 38
Messages command 60
Messages menu, TCINST 200
Modal Help

TCINST option 212
MODE command (DOS) 7
Mode for Display menu, TCINST 197, 210
Model option, TCINST 199
models, memory See memory models
Modify editor macro 226
module names, TLIB 172
monitors See screens
dual 7, 27, 73
number of lines 75
setting default 210
Monochrome option, TCINST 211
More ANSI Violations dialog box 62
More Frequent Errors dialog box 63
More options, TCINST 201
mouse
buttons
switching 78
choosing commands with 9, 18
double click 207
double-click speed 78
left-handed
support for 78
reverse 207
reversing buttons 78
right button 207
right button action 78
selecting text with 27
support for 5
Mouse Double Click option 78
Mouse Double Click option, TCINST 207
Mouse menu, TCINST 207
MoveBlock editor macro 226
MoveToBlockBeg editor macro 227
MoveToBlockBegRaw editor macro 227
MoveToBlockEnd editor macro 227
MoveToBlockEndRaw editor macro 227
MoveToMark editor macro 227
MoveToPrevPos editor macro 227
moving text See editing, block operations
multi-source programs See projects
multiple files See projects
multiple listings
command-line compiler options
#define 119
include and library 131

macro definition 119
–mx options (memory models) 118

N

–n MAKE option (print commands but don’t execute) 141
–n TCC option (.OBJ and .ASM directory) 130
–N TCC option (stack overflow logic) 121
/n TLINK option (ignore default libraries) 184
$NAME transfer macro 68
Names
command 63
names See identifiers
Names menu, TCINST 202
neste
comments 124
toggle 200
delimiters See delimiters
Nested Comments option 59, 111
New command 24, 106
New Value field 43
New Window option 76
Next command 80
hot key 11, 12
macro 227
Next Error command 33
hot key 13
macro 227
NextError editor macro 227
NextWindow editor macro 227
.NOAUTODEPEND MAKE directive 160
.NOIGNORE MAKE directive 160
NONAME file name 24
nondirectional delimiters See delimiters
nondirectional pair matching 109
nonfatal errors See errors
.NOSILENT MAKE directive 160
.NOSWAP MAKE directive 160
$NOSWAP transfer macro 69
Notes command 81
null character See characters, null
–num MAKE command 146
numbers See also floating point; integers
column 67
decimal 74
format specifier 45
hexadecimal 74
constants
  too large 125, 201
  format specifier 45
line 67
octal
  constants
  too large 125
real See floating point
numeric coprocessors See also floating point
emulating 120
generating code for 120, 121
inline instructions 55, 120
libraries
  TLINK and 181
Turbo C++ options 199
using 199

-0 TCC option (object files) 128
-O TCC option (optimization) 123
/o TLINK option (overlays) 186
.OBJ files
  compiling 128
  creating 38
directory 205
  debugging information 56
creating 173
dependencies 70
directories 75, 130
duplicate symbols in 185
directory 205
creating 173
  TLINK and 169
  libraries
  advantages of using 170
  creating 173
  TLIB and 169
  line numbers in 122
  object files See .OBJ files
object-oriented programming See C++
objects See C++
object-oriented programming See C++
object-oriented programming See C++
object-oriented programming See C++
octal numbers See numbers, octal
OK button 18
online help See help
OOP See C++
Open command 22, 106
  hot key 11, 12
Open Project
  command 49
OpenFile editor macro 227
operations (TLIB option) 171
operators
  MAKE 146, 163
  precedence 164
  precedence See precedence
Optimal Fill option 77
Optimal Fill option, TCINST 206
optimization 58
  command-line compiler options 118, 123
  fast floating point 56
  for speed or size 58
  jump 200
  register 200
  selecting 200
Optimization menu 123
Optimizations
  command 57
dialog box 58
Optimizations menu, TCINST 200
Optimize For option, TCINST 200
options See command-line compiler, options;
advanced environment, command-line
options; MAKE (program manager), options;
TLIB (librarian), options; TLINK (linker),
options
Options menu 52
  command-line compiler and 114
  macro 227
Options menu, TCINST 197, 198
OptionsMenu editor macro 227
ordinals, inspecting 40
Origin option, TCINST 197
Out-Line Inline Functions option
Out-of-line Inline Functions option, TCINST
Out-of-line Inline Functions option, TCINST
output
  capturing 68
directory 205
to DOS
  viewing from IDE 80, 81
  User Screen 81
Output command 80
Output Directory
  input box 75
Output Directory option, TCINST 205
Output window
  TCINST option 212
Overlay EXE option, TCINST 203
Overlay Support option, TCINST 198
overlays
default support 198
.EXE files 203
projects and 51
supporting 53
TLINK and 186
toggling 72
Override Options dialog box 51
Overwrite Mode 77

P
/p IDE option (EGA palette) 8
-p TCC option (Pascal conventions) 121
/P TLIB option (page size) 174
page size (libraries) 174
PageDown editor macro 227
PageScreenDown editor macro macro 227
PageScreenUp editor macro 227
PageUp editor macro 227
PaintScreen editor macro 228
pair matching 108-112
  angle brackets 108
  backward 109
  braces 108
  brackets 108
  commands 108
  comment delimiters 108, 110, 111
directional 109
double quotes 108
examples 111
forward 109
nested expressions 108
nondirectional 109
parentheses 108
rules 109
  single quotes 108
parameter-passing sequence, Pascal 121
parameters See arguments
Pascal
calling convention 55
Calling Sequence 199
identifiers of type 121
parameter-passing sequence 121
Paste command 28, 105
  hot key 12
  macro 224
pasting See edit, copy and paste
path names in Directories dialog box 75
.PIPE directive (MAKE) 160
place markers (editor) 106
pointers
comparisons
  non-portable 200
conversion
  non-portable 200
suspicious 200
format specifier 45
inspecting values 40
memory regions 45
mixing 200
suspicious conversion 125
polymorphism See C++
pop-up menus 9, See also menus
Pop-Up Menus option, TCINST 212
Portability
dialog box 61
Portability options, TCINST 200
portability warnings 61, 126
precedence
  command-line compiler options 134
MAKE operators 164
TLIB commands 172
Preferences
dialog box 75
Preferences dialog box 75
Preferences options, TCINST 205
PrevError editor macro 228
Previous Error command 33
  hot key 13
  macro 228
Previous Topic command 84
  hot key 12
  macro 226
Print Block command 105
Print command 25
Print File command 105
.PRJ files See projects
$PRJNAME transfer macro 67
projects
notes 100
procedures See functions
Program Heap Size option, TCINST 203
program manager (MAKE) See MAKE
Program Reset command 35
   hot key 13
programs
   C++  See C++
   capturing output 68
   ending 33
   file name 67
   heap size 74
   memory assignments 69
   multi-source  See projects
   rebuilding 34, 38
   resetting 35
   macro 228
running 34
   arguments for 37
   macro 228
   to cursor 35
   Trace Into 35
transfer
   list 96
transferring to external from Turbo C++ 64
Project
   command 81
   menu 49
project
   excluding from 51
Project File
   dialog box 49
Project-Make
   macro 226
Project Manager 34, 87-100,  See also projects
   closing projects 50
   Include files and 51
Project Name
   command 66
Project Notes window 100
Project option 76
projects 87-100,  See also Project Manager
   autodependency checking 70
   automatic dependency checking and 94
   building 87
   changing 86
   closing 50
   default 86
   desktop files and 86, 84-86
   directories 91
   directory 85
error tracking 91, 92
   .EXE file names and 38
files
   adding 89
   command-line options and 51
   deleting 89, 90
   file name transfer macro 67
   include 89
   information 95
   list 89
   modifying 196
   options 89
   out of date 94
   viewing 99
IDE configuration files and 84
   include files 89
   information in 87
   libraries and
      overriding 97
         math libraries and 97
loading 85
   makes and 93
   making
      hot key for 92
   managing 81
   meaning of 50
   naming 88
   new 89
   notes 81
   overlays and 51
   saving 90
   translator option 51
   translators  See also Transfer
      default 95
      example 96
      multiple 94
      specifying 95
Prompt on Replace option, TCINST 198
$PROMPT transfer macro 69
pseudo variables, register
   using as identifiers 124
Public option
   C++ Virtual tables 57
Pull-down Menu option, TCINST 212
pull-down menus  See menus
Q
-\text{Q} TCC option (expanded and extended memory) 129
Quit
command 8, 27, 106
editor macro 228
Quit option, TCINST 197, 213

R
-\text{r} TCC option (register variables) 123
/rx IDE option (RAM disk) 7
radio buttons 19
RAM disk
    integrated environment and 7
RAM disks
    expanded and extended memory and 129
-\text{rd} option (register variables) 123
ReadBlock editor macro 228
real numbers \textit{See} floating point
rebuilding libraries 122
redirecting program output 68
redirection
    DOS 198
    operators
        MAKE 146
register (keyword)
    Register Variables option and 58
Register command 81
Register Optimization option 58
Register Optimization option, TCINST 200
Register Variables option 58
Register Variables option, TCINST 200
registers
    DS (data segment pointer) 56
    pseudovariables
        using as identifiers 124
    reusing 58
    SS (stack segment pointer) 56
    variables 123
        suppressed 123
        toggle 123, 200
    windows 81
Regular Expression option, TCINST 197
remove (TLIB action) 173
Remove All Watches command 47
Remove Messages command 38, 93
Repaint Desktop command 22
RepeatSearch editor macro 228
Replace
    command 31, 108
    dialog box 32, 108
    editor macro 228
replace (TLIB action) 173
Replace command
    hot key 12
ResetProgram editor macro 228
resetting programs 35
resident utilities
    expanded and extended memory and 129
    resize corner 15
    resize corners 16
Resize Windows option, TCINST 197
response files
    defined 117, 178
    TLIB 173
    TLINK and 178
Restore Line command 28, 102, 106
RestoreLine editor macro 228
restrictions, TLINK 187
Result field 43
Reverse Mouse Buttons option 78
Reverse Mouse option, TCINST 207
Right Mouse Button option 78
Right Mouse Button option, TCINST 207
RightOfLine editor macro 228
Ritchie, Dennis \textit{See} Kernighan and Ritchie
Run
    command 34
    menu 33
    macro 228
Run command
    hot key 13
Run menu, TCINST 197, 198
RunMenu editor macro 228
running programs 34
RunProgram editor macro 228
RunTohere editor macro 228

S
-\text{s} MAKE option (don’t print commands) 141
-\text{S} MAKE option (swap MAKE out of memory) 141
S TCC option (produce .ASM but don’t assemble) 128
/s TLINK option (detailed segment map file) 178
sample programs
  copying from Help window 28
$SAVE ALL transfer macro 69
Save All command 24
Save As
  command 24
$SAVE CUR transfer macro 69
Save command 24, 79, 106
  hot key 11, 12
Save Configuration menu, TCINST 213
Save File As dialog box 24
Save Old Messages option 76
Save Old Messages option, TCINST 205
$SAVE PROMPT transfer macro 69
SaveFile editor macro 228
Scope option, TCINST 197
screen
  Color 210
Screen Size
  option 75
Screen Size menu, TCINST 205
screens
  43/50-Line Display 205
  25-Line Display 205
  Black and White 211
composite
  customizing Turbo C++ for 210
Default 210
editor macros
  cursor position 223, 224
  movement 222
  painting 225, 228
  position relative to cursor 225
  refreshing 229
  scrolling 222, 227, 228, 229
LCD
  integrated environment option 8
LCD or Composite 211
Monochrome 211
number of lines 75
painting
  macros for 223
  repainting 22
size
  setting default 205
swapping 203
top of
  macro 230
two
  using 7
scroll bars 15
ScrollDown editor macro 228
scrolling windows 15
ScrollScreenDown editor macro 229
ScrollScreenUp editor macro 229
ScrollUp editor macro 229
Search Again command 32, 107
  hot key 12
search and replace 107-108, See also searching
Search menu 29, 107-108
  macro 229
Search menu, TCINST 196, 197
searching
  and replacing text 107, 107-108
  direction 31
  error and warning messages 33
  functions 33
  in list boxes 83
  include files 131
  libraries 131, 132
  origin 31
  regular expressions 30
  repeating 32
  and replacing text 31
  scope of 31
  search and replace 31
searching and replacing text 107-108
SearchMenu editor macro 229
Segment Names dialog box 64
segment-naming control
  command-line compiler options 118, 127
segments
  controlling 127
  initializing 71
  toggle 202
map of
  ACBP field and 183
  TLINK and 182
names 63
uninitialized
TLINK and 184

semicolons (;) in directory path names 75
SetBlockBeg editor macro 229
SetBlockEnd editor macro 229
SetBreakpoint editor macro 229
SetInsertMode editor macro 229
SetMark editor macro 229
SetPrevPos editor macro 229
shortcuts See hot keys
Show Clipboard command 29
.SILENT MAKE directive 160
Size/Move command 79
Smart option
  C++ Virtual tables 57
smart screen swapping 203
SmartRefreshScreen editor macro 229
snow 210
software See programs
Source
  command 59, 111
Source Debugging command 34
  and Trace Into command 36
source files
  .ASM 113
  command-line compiler options 118
  multiple See projects
  separately compiled 170
source-level debugger See Turbo Debugger
Source menu, TCINST 200
Source Options dialog box 59, 111
Source Tracking option 76
Source Tracking option, TCINST 205
Source Tracking options 93
spaces vs. tabs 77
spreadsheets See Turbo Calc
SS register (stack segment pointer) 56
stack
  Call Stack command 44
  overflow 54, 121
    testing 199
  standard frame
    generating 121
    TCINST and 199
  warnings 72
  toggle 203
Stack Warning option, TCINST 203

standalone debugging information 72
standalone utilities 137, See also MAKE
  (program manager); TLIB (librarian); TLINK (linker); TOUC
  configuration file converter 135
standard library files See libraries
standard stack frame
  generating 121
Standard Stack Frame option 44, 53
Standard Stack Frame option, TCINST 199
startup code (TLINK) 180
startup modules for memory models 181
status line 17
Status Line option, TCINST 212
Step editor macro 229
Step Over command 36
  hot key 11, 13
  macro 229
strings
  duplicate
    merging 53, 198
  format specifier 45
  inserting
    macro 226
  literal
    merging 120
structures
  ANSI violations 125
  format specifier 45
  inspecting 41
  undefined 125
  zero length 125
suppressing load operations 123
.SWAP MAKE directive 160
swapping
  displays 73
  to User Screen 69
SwapPrevPos editor macro 229
switches See command-line compiler, options;
  integrated environment, options
symbolic
  constants See macros
  debugger See Turbo Debugger
symbols
  action See TLIB
  duplicate 72
    warning 203
warning (TLINK) 184
syntax
  errors
    project files 91, 92
  tracking 205
IDE command line 6
MAKE 139
TLIB 171
TLINK 177
system
  editor macros for 223
System menu
    macro 229
System menu = 8
SystemMenu editor macro 229

T
/t TLINK option (default to .COM) 185
/t TLINK option (generate .COM file) 178, 180
Tab Size option 77
Tab Size option, TCINST 206
Table of Contents command
    macro 226
tables
    virtual 199
tabs
    characters
      printing 25
      size of 77
      spaces vs. 77
    using in the editor 77
TASM  See Turbo Assembler
TASM2MSG.EXE 68
$TASM transfer macro 69
TC.EXE  See integrated environment
TCC.EXE  See command-line compiler
TCCNVNVT.EXE
    configuration files and 135
TCCNVNVT.TC
    converting to TURBOC.CFG 135
TCCCONFIG.TC  See configuration files, IDE
TCCCONFIG.TC. modifying 196
TCDEF.DPR files 86
TCDEF.DSK files 86
TCINST 195-213, See also TCINST menu and
    command names
      black-and-white option 196
colors
      changing 196, 212
default command set 13
text See also editing
    blocks  See editing, block operations
    copy and paste 28
    cutting 28
    deleting 29
      macro 225
    entering 102
      in dialog boxes 19
    inserting vs. overwriting 77
    pasting 28
    screen display of 75
    screen mode  See screens, operating modes
    selecting 27
      Help window 82
32-bit code 186
/3 TLINK option (80386 32-bit code) 186
tilde (~) in transfer program names 65
tile command 80
title bars 15
TLIB (librarian) 169-176
    action symbols 172-173
    capabilities 170
    examples 176
    libraries
      creating 171
    module names 172
    operations 172
      precedence 172
    options
      case sensitivity (/c) 171, 175
      /E 171, 174
      extended dictionary (/e) 171, 174
$EXT 68
file names 66, 68
glossary of 67
hot keys for 65
how expanded 66
instruction 66
$LINE 67
$MEM 68
$NAME 68
$NOSWAP 69
$PRJNAME 67
$PROMPT 69
$SAVE ALL 69
$SAVE CUR 69
$SAVE PROMPT 69
$TASM 69
transfer programs
list 96
Translator option 51, 66
translators See projects, translators
Treat enums as ints option 56
Treat enums as ints option, TCINST 199
Turbo Assembler
capturing messages 68
default 128
invoking 116
$TASM macro 69
TLINK and 186
Turbo C++ and 113
Turbo C++ command-line compiler and 117
Turbo C++ See also C++; C language; keywords C and 121
calling convention 55
keywords 200
as identifiers 59, 124
project files and 94
quitting 8, 27
starting up 5
transferring from 64
Turbo Debugger
described 122
Turbo Editor Macro Language compiler See editor, macro language (TEML)
TURBOCFG 133
converting to TCCNVT.TC 135
25-line display 75, 205
typefaces used in these books 2
types
documenting 43

U
-U MAKE option (undefine) 141
-U TCC option (undefine) 119
-u TCC option (underscores) 121
unconditional breakpoints See breakpoints
!undef MAKE directive 165
underbars See underscores
underscores 121

generating automatically 56, 121, 199
unindent mode
default 206
uninitialized data segment See data segment
unions
format specifier 45
inspecting 41
UNIX
keywords 59
using 124
porting Turbo C++ files to 124
UNIX V
keywords 200
unmatched delimiters 111
Unsigned Characters option 53
Unsigned Characters option, TCINST 199
Use C++ Compiler option, TCINST 199
Use Tab Character option 77
Use Tab Character option, TCINST 206
User Screen 69
hot key 12
viewing
macro 230
User Screen command 81
user-specified library files 131
utilities
resident
expanded and extended memory and 129

V
-v option (debugging information) 184
-v TCC option (debugging information) 122
/v TLINK option (debugging information) 186
variable argument list 121
variables See also scope
  automatic
    word-aligning 120
debugging 43
inspecting values of 39
register 58, 123
Verbatim option, TCINST 209
version number information 21
-vi option (C++ inline functions) 122
Video Graphics Array Adapter (VGA) 76
screen size 205
ViewCallStack editor macro 230
ViewFullOutput editor macro 230
virtual tables 57, 199
void functions
  TCINST option 200

W
-W MAKE option (save options) 141
-wxxx TCC options (warnings) 125
Warn Duplicate Symbols option, TCINST 203
warnings See also errors
  ANSI Violations 61
C++ 62, 126, 201
  command-line options 125-127
displaying 200
duplicate symbols 203
enabling and disabling 125
frequent errors 62, 126
options 125-127
portability 61, 126
stack 203
stopping on n 60
  setting n 200
TLINK
  defined 188
  list 192
types of 60
Warnings : Stop After option, TCINST 200
Watch window
  TCINST option 212
Watches command 45
watchpoints See debugging
whole-word searching 30
Whole Words Only option, TCINST 197
wildcards 30
  DOS 23
GREP 30
MAKE and 150
TOUCH and 193
Window menu 79
window number See windows, window number
WindowList editor macro 230
windows
  active 16
    defined 14
cascading 80
Clipboard 29
closed 81
closing 15, 16, 21, 80
current
  macro 224
Edit See Edit, window elements of 14
Help See Help, windows
Inspector 39
jumping to
  macro 225
List 81
macros 223
  list of open 230
menu 79
Message 38, 80
moving 17, 79
next 80
  macro 227
Notes 81
open 81
opening 16, 79
Output 80
position
  hot key 12
Project 81
Register 81
reopening 79
resizing 16, 17, 79
scrolling 15
size
  hot key 12
source tracking 76
swapping in debug mode 73
dual monitors and 73
tiling 80
title bar 15
User Screen 81
Watch 81
window number 16
zooming 15, 16, 17, 79
macro 230
Windows menu
macro 230
WindowsMenu editor macro 230
word aligning
integers 120
Word Alignment option 53
Word Alignment option, TCINST 198
WordLeft editor macro 230
WordRight editor macro 230
Wordstar-like option, TCINST 209
WriteBlock editor macro 230
-wxxx TCC options (warnings) 125-127

X
/x IDE option (expanded memory) 7
/x TLINK option (map files) 178

Y
-y TCC option (line numbers) 122
/y TLINK option (expanded and extended memory) 187

Z
-Z TCC option (aliasing) 123
zoom box 15, 16
Zoom command 79
hot key 11, 12
ZoomWindow editor macro 230
-zX options (code and data segments) 127, 128