Turbo Pascal® for Windows

User's Guide

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

R1

10 9 8 7 6 5 4 3

C O N T E N T S

T A B L E S

F I G U R E S

INTRODUCTION

Turbo Pascal for Windows is designed for all types of users who want to develop applications that run under Microsoft Windows. Whether you are a beginner seeking to enter the world of programming, or an advanced developer looking for a better and easier way to create applications for Windows, Turbo Pascal offers you a rich programming environment that makes software development more productive—and more fun. Using Pascal's structured, high-level language you can write programs for any type or size of application.

Turbo Pascal for Windows includes

- a Windows-hosted integrated development environment (IDE) based on Windows Multiple Document Interface (MDI).
- ObjectWindows, a Pascal object-oriented class library that simplifies the task of programming for Windows.
- interactive tools to create and edit the icons, cursors, dialog boxes, menus, and accelerators you use in your programs. You add these resources to your program by including them in your source code with the new **$R** compiler directive.
- Turbo Debugger for Windows, a powerful debugger for Windows applications that you can run from the IDE.
- support for the creation of Dynamic Link Libraries (DLLs) and the capability to debug them with Turbo Debugger for Windows.
- Dynamic Method Tables, which decrease the memory requirements of your executable programs when you program with objects.
- full access to all the functions and procedures that make up the Windows Application Programming Interface (API) in the *WinProcs* unit.

- the *WinCrt* unit, which provides a fully functioning window for simple text-based programs so you can be productive in the Windows environment almost instantly.
- optional use of null-terminated strings.
- a built-in assembler for your inline assembly routines.
- complete online help for the IDE, the Pascal language, ObjectWindows, and the Windows API.

Turbo Pascal offers you a new, better way to program for Windows. While exploring the opportunities that await you, remember that this is still Turbo Pascal, the quick and efficient Pascal compiler that is already the world's standard.

The Turbo Pascal for Windows manuals

Turbo Pascal for Windows comes with six manuals, each with a different purpose. Briefly, here's what each contains:

The *User's Guide* (this book) explains how to install, learn, and use Turbo Pascal's integrated environment and command-line compilers. It also includes the basics of programming in Turbo Pascal, as well as more advanced topics like object-oriented programming and the management of large projects.

The *Programmer's Guide* is a reference guide to technical aspects of Turbo Pascal, describing in detail the definition of the language, the contents of the standard libraries and how they are implemented in Turbo Pascal, and the use of Turbo Pascal with assembly language. This volume also contains explanations of all compiler directives and error messages used by Turbo Pascal, as well as an alphabetical reference to all the standard procedures and functions in the Turbo Pascal run-time library.

In the *Windows Programming Guide* you'll learn how to develop applications for Windows using ObjectWindows.

In the *Windows Reference Guide* you'll find a reference to all the constants, styles, messages, and API functions in the Windows interface, as well as all the objects in the ObjectWindows library.

You'll learn how to create your resources in the *Whitewater Resource Toolkit*.

Finally, the *Turbo Debugger for Windows User's Guide* gives you all the details about debugging your Window applications with Turbo Debugger for Windows.

Installing Turbo Pascal

Turbo Pascal comes with an automated installation program called INSTALL. You should use INSTALL to load Turbo Pascal onto your system, as it will ensure that you get all the files you need into the places where you need them. INSTALL will automatically create directories and Program Manager groups and copy files from the distribution disks to your hard disk.

If you have Windows in your path, you can start INSTALL from the DOS command line.

1. Insert your Turbo Pascal installation disk into drive A.
2. Type `WIN A:INSTALL` and press *Enter*.
3. Set options in the dialog box that appears.
4. Choose Install and INSTALL begins copying files.

If Windows is not in your path, you can start Windows first and follow these steps:

1. Insert your Turbo Pascal installation disk into drive A.
2. Choose File | Run in the Windows Program Manager.
3. Type `A:INSTALL` and choose OK.
4. Set options in the dialog box that appears.
5. Choose Install and INSTALL begins copying files.

Starting Turbo Pascal

To start Turbo Pascal, double-click the Turbo Pascal icon in the Program Manager or select it with your keyboard and press *Enter*.

You can also start Turbo Pascal from the DOS prompt by typing `WIN TPW.EXE` and even specify a configuration file; see Chapter 6 for more information.

Customizing Turbo Pascal

The integrated development environment (IDE) allows you to customize the way Turbo Pascal behaves by selecting options and preferences without exiting the program to use external utilities.

The IDE saves the options you have set so that the IDE is just as you left it the next time you start up Turbo Pascal. If you don't want this to happen, choose Preferences from the Options menu to open the Preferences dialog box and uncheck Auto Save options, Configuration and Desktop. Choose OK.

The README file

The README file contains last-minute information that may not be in these manuals. It also lists every file on the distribution disks, with a brief description of what each one contains.

Here's how to access the README file:

1. Insert your Turbo Pascal installation disk into drive A.
2. Type `A:` and press *Enter*.
3. Type `README` and press *Enter*. Once you're in README, use the *PgUp*, *PgDn*, ↑ and ↓ keys to scroll through the file.
4. Press *Esc* to exit.

After you've installed Turbo Pascal, you can open README in an edit window by following these steps:

1. Start Turbo Pascal.
2. Choose Open from the File menu. Type `README.` (don't forget the .) in the input box and choose OK. The README file opens in an edit window.
3. When you're done with the README file, choose Close from the File menu or press *Alt+F4* to exit Turbo Pascal.

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 on-screen or in a program. It is also used for anything you must type (such as `TPW` to start up Turbo Pascal).
[ ]	Square brackets in text or DOS command lines enclose optional items that depend on your system. *Text of this sort should not be typed verbatim.*
Boldface	This typeface is used in text for Turbo Pascal reserved words, for compiler directives {**$I-**} and for command-line options (**/A**).
Italics	Italics indicate 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, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."
	This icon indicates keyboard actions.
	This icon indicates mouse actions.

How to contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type `GO BOR` from the main CompuServe menu and choose "Borland Programming Forum A (Turbo Pascal)" from the Borland main menu. Leave your questions or comments there for the support staff to process.

If you prefer, write a letter with your comments and send it to

Borland International
Technical Support Department—Turbo Pascal
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001, USA

You can also telephone our Technical Support department between 6 a.m. and 5 p.m. Pacific time at (408) 438-5300. Please have the following information handy before you call:

1. Product name and serial number on your original distribution disk. Please have your serial number ready, or we won't be able to process your call.
2. Product version number. To find the version number for Turbo Pascal, choose About Turbo Pascal from the Help menu.
3. Computer brand, model, and the brands and model numbers of any additional hardware.
4. Operating system and version number. (Type `VER` at the DOS prompt.)
5. Windows version number and mode. To find this information, choose About Program Manager from the Windows Program Manager's Help menu.
6. Contents of you AUTOEXEC.BAT file.
7. Contents of your CONFIG.SYS file.

C H A P T E R

1

Learning the IDE

Turbo Pascal for Windows is more than just a fast, efficient Pascal compiler for Windows; it is an easy-to-learn and easy-to-use integrated development environment (for short, we call it the IDE). With Turbo Pascal for Windows you don't need to use a separate editor, compiler, and linker to create and run your Pascal programs. All these features are built into Turbo Pascal, and they are all accessible from the IDE. You can even use the IDE to start up Borland's powerful Turbo Debugger for Windows.

You can begin building your first Turbo Pascal for Windows program using the compiler built into the IDE. By the end of this chapter, you'll have learned your way around the development environment, written and saved two small programs, and learned some basic programming skills.

Online context-sensitive help is only a keystroke (or a mouse click) away. You can get help at any point (except when *your* program has control) by pressing the shortcut *F1*. The Help menu (*Alt+H*) provides you with an index to the help system (*Shift+F1*), specific language help about the item your cursor rests on with Topic Search (*Ctrl+F1*), help on using the help system itself, and detailed help about Turbo Pascal, ObjectWindows, and the Microsoft Windows Application Programming Interface (API).

Because Turbo Pascal runs under Windows, we assume you have some knowledge about using Windows. If you are comfortable with Windows, you will feel right at home with Turbo Pascal.

If you want more details about the IDE, look at Chapter 6, "The IDE reference."

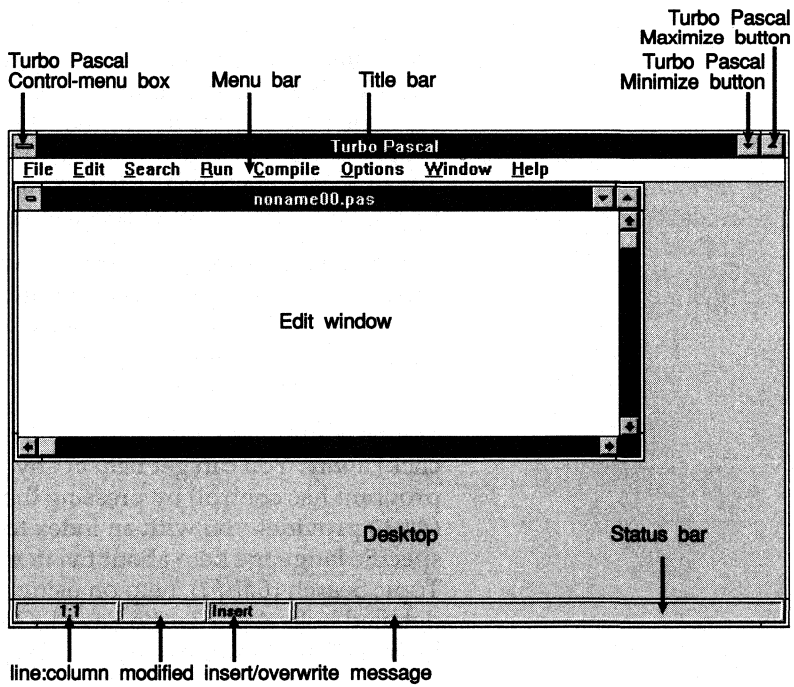
Starting up Turbo Pascal

Starting up Turbo Pascal for Windows is easy. From within Windows Program Manager, double-click the Turbo Pascal icon with your mouse. If you are using your keyboard, select the Turbo Pascal icon with your cursor keys and press *Enter*.

When Turbo Pascal starts, you'll see the About Turbo Pascal box that displays the Borland copyright notice and version number. Choose OK to put the box away.

You will see one edit window titled `NONAME00.PAS` on the Turbo Pascal desktop. This is how your screen should look:

Figure 1.1
The starting screen



There are three main parts to the IDE: the title bar and menu bar at the top, the desktop, and the status bar at the bottom. *Edit windows*, areas where you create and edit your programs, open on the Turbo Pascal desktop.

The menu bar and menus

Beneath the Turbo Pascal title bar is the *menu bar*, which you use to access all the menus. The menu bar is always visible as long as you are working in Turbo Pascal.

To display a menu with a mouse, click the menu title on the menu bar. To put the menu, click another part of the screen.

There are two ways to display a menu using your keyboard:

- Press *Alt* to make the menu bar active. Use your arrow keys to move to the menu title you want and press *Enter* or ↓.
- Press *Alt* plus the first letter of the title of the menu you want to display. Note that the first letter of each menu title on the menu bar is underlined. For example, pressing *Alt+E* displays the Edit menu.

To put away a menu without selecting a command, press *Esc*.

Each menu lists a series of commands. If a menu command is followed by an ellipsis mark (...), choosing the command displays a dialog box. A command without an ellipsis mark indicates that once you choose it, the action occurs.

At times menu commands will appear dim and, when you choose them, nothing occurs. Turbo Pascal does this when choosing a command doesn't make sense in your current context. For example, if you don't have a block selected in your current edit window, you won't be able to cut, copy, or clear text because you haven't told the editor what text you want cut, copied, or cleared. The Cut, Copy, and Clear commands will therefore be dimmed on the Edit menu. Once you select text in your edit window, however, you can choose these commands.

Using a mouse, click the menu command you want. If you decide you don't want to select a menu command, click some part of your screen other than the displayed menu and the menu will disappear.

Using the keyboard, select the command you want with the arrow keys. Press *Enter*. To put away a menu without selecting a command, press *Esc*. Press *Esc* again to leave the menu bar and return to the active edit window.

Shortcuts

Turbo Pascal offers a number of quick ways to choose menu commands. For example, if you are a mouse user, you can combine the two-step process of displaying a menu and selecting a command into one step by simply dragging from the menu title down to the menu commands and releasing the mouse button when the command you want is selected. (If you change your mind, just drag off the menu; no command will be selected.)

From the keyboard, you can use a number of keyboard shortcuts (or *hot keys*) to access the menu bar and choose commands. Here's a list of the shortcuts available:

To accomplish this:	Do this...
Display the desired menu	Press *Alt* plus the underlined letter of the menu title. For the Turbo Pascal Control menu, press *Alt+Spacebar*.
Carry out the command or display a dialog box	Once a menu is displayed, press the underlined letter of the command you want.
Carry out the command	Type the keystrokes next to a menu command.

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 the command on the Edit menu.

Many menu items have corresponding shortcuts: one- or two-key hot keys that immediately activate that command or dialog box. The following tables list the Turbo Pascal hot keys.

Table 1.1
Menu title hot keys

Key(s)	Menu Item	Function
Alt+Spacebar	Control menu	Displays the Turbo Pascal Control menu.
Alt+Hyphen	Window Control menu	Displays the active window Control menu.
Alt+F	File menu	Displays the File menu.
Alt+E	Edit menu	Displays the Edit menu.
Alt+S	Search menu	Displays the Search menu.

Table 1.1: Menu title hot keys (continued)

Key(s)	Menu Item	Function
Alt+R	Run menu	Displays the Run menu.
Alt+C	Compile menu	Displays the Compile menu.
Alt+O	Options menu	Displays the Options menu.
Alt+W	Window menu	Displays the Window menu.
Alt+H	Help menu	Displays the Help menu.

The Turbo Pascal for Windows editor has two command sets: CUA and Alternate. When you run Turbo Pascal for the first time, the CUA command set is active. A few of the menu shortcuts you see on the menus vary depending on which command set is in effect. See Chapter 7 to help you decide which command set you want to use and how to switch command sets.

Table 1.2
Menu command hot keys

Key(s)	Menu Item	Function
Alt+F4[1]	Control | Close/ File | Exit	Closes the Turbo Pascal desktop.
Alt+X[2]	File | Exit	Closes the Turbo Pascal desktop.
Ctrl+Esc	Control | Switch To	Transfers control to Task List.
Ctrl+F4	Edit Window Control | Close	Closes the active edit window.
Ctrl+F6 or *Ctrl+Tab*	Edit Window Control | Next	Transfers control to next open window.
F3[2]	File | Open	Opens a file.
F2[2]	File | Save	Saves the file in the active window.
Alt+BkSpace	Undo	Restores the current window to the way it was before the most recent edit or cursor movement.
Shift+Del	Edit | Cut	Cuts the current selection into the Clipboard.
Ctrl+Ins	Edit | Copy	Copies the current selection into the Clipboard.
Shift+Ins	Edit | Paste	Pastes clipboard contents into the active window.
Ctrl+Del	Edit | Clear	Cuts current selection without copying to the Clipboard.

Table 1.2: Menu command hot keys (continued)

Key(s)	Menu Item	Function
F3[1]	Search | Search Again	Repeats last Find or Replace.
Ctrl+F9	Run | Run	Checks that program in active window is up-to-date, loads it, and runs it.
Alt+F9	Compile | Compile	Compiles the file in the active window.
F9	Compile | Make	Compiles files that need updating.
Shift+F5	Window | Tile	Tiles Turbo Pascal windows.
Shift+F4	Window | Cascade	Cascades Turbo Pascal windows.

[1]A command in the CUA command set; see Chapter 7.

[2]A command in the Alternate command set; see Chapter 7.

Table 1.3
Help hot keys

Key(s)	Menu Item	Function
F1		Displays context-sensitive help.
Shift+F1	Help | Index	Displays the help index.
Ctrl+F1	Help | Topic Search	Displays language help on a specific item

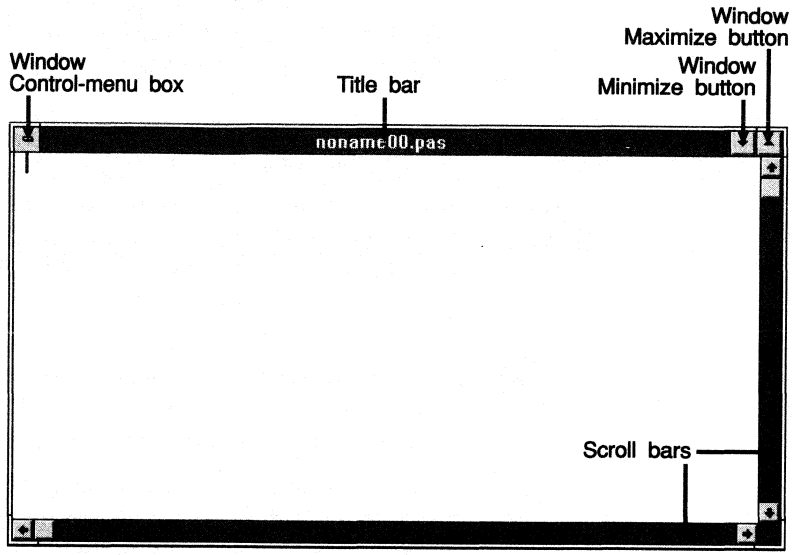
Managing Turbo Pascal windows

Most of what you see and do in Turbo Pascal happens in an edit window. A window is a screen area that you can open, move, resize, cascade, tile, overlap, close, or shrink into an icon.

You can have up to 32 windows open in Turbo Pascal (memory allowing), but only one window can be *active* at any time. The active window is the one you're currently working in. Any command you choose or text you type usually applies only to the active window. (If you have the same file open in more than one window, editing actions will apply to all windows with the same file.)

This is how a typical window looks:

Figure 1.2
An edit window



Turbo Pascal edit windows have these things in common:

- a title bar
- a window Control-menu box
- scroll bars
- a Minimize button
- a Maximize button

If a window has been maximized (enlarged to take up the whole Turbo Pascal desktop), it will also have a *Restore* button.

The *title bar*, the topmost horizontal bar of a window, contains the name of the file in the window, the *Control-menu button*, and the *Minimize* and *Maximize* buttons.

***Shortcuts**: Alt+Hyphen (-) displays the active window Control menu.*

A window's Control-menu box displays the Control menu for the active window if you click it once; it closes the window if you rapidly click it twice and the window is not maximized. (You can also choose Control | Close or press *Ctrl+F4*.)

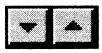
Scroll bars let both mouse and keyboard users see how far into the file they've gone.

Scroll bars are the horizontal and vertical bars at the bottom and right side of the window. You can use these bars with a mouse to scroll the contents of a window. Click the scroll arrow (a button with an arrow) at either end of a scroll bar to scroll one line at a time. (Keep the mouse button depressed to scroll continuously.) You can click the shaded area on either side of the *scroll box* to

scroll a line at a time. Finally, you can drag the scroll box to anywhere on the bar to quickly move to a spot in the window relative to the position of the scroll box.

Resizing a window

The *Minimize* and the *Maximize buttons* are on the right side of the title bar. The Minimize button, a single-headed arrow pointing down, shrinks your active window into an icon on the Turbo Pascal desktop. The Maximize button, a single-headed arrow pointing up, enlarges the active window so that it fills up the Turbo Pascal desktop completely.



If the active window has been maximized, it will have a *Restore* button instead of a *Minimize* or *Maximize* button. The Restore button is in the upper right corner. Clicking the Restore button resizes the active window to its normal size.

You can reshape and resize your window by moving the mouse cursor to a window border. When the cursor changes from a single-headed arrow to a double-headed arrow, press and hold the mouse button while dragging the window border. When the border is where you want it, release the mouse button.

To move a window, click the title bar of the window and hold the mouse button down while you drag the window around the desktop. When you are satisfied with the position of the window, release the mouse button.

You can resize your windows from your keyboard by using the commands on the active window. Press *Alt+Hyphen* (-) to display the Control menu. The Minimize and Maximize commands act just like the Minimize and Maximize buttons. The Restore command returns the window to its previous size. Choosing Size lets you use your arrow keys to move the window borders to expand or shrink the window. Once you select Size, press your arrow keys in the direction you want a window border to move. When you are satisfied with the location of the border, press *Enter.*

In the same manner, you can move a window with your keyboard. Select Move from the window Control menu. Use your cursor keys to move the window around and then press *Enter* when you're done.

Working with a window icon

Once you shrink a window so that it appears as an icon on the Turbo Pascal desktop, you can still make it the active window, move it, or close it, as well as restore it to its previous size. With your mouse you can click the icon once to display the window Control menu. Then you can select from the menu's Restore, Move, Maximize, Close, or Next commands.

You can double-click the icon to quickly restore it, or click and drag the icon to rearrange it on the Turbo Pascal desktop.

If you are managing your windows with a keyboard, make the window icon your active window (press *Ctrl+F6* or *Ctrl+Tab* until the icon is selected) and press *Alt+Hyphen* (-) to display the window Control menu. Select from the Restore, Move, Maximize, Close, or Next commands.

Window management summary

The following table summarizes how you handle windows in Turbo Pascal for Windows. Note that you don't need a mouse to perform these actions—a keyboard works just fine.

Table 1.4
Manipulating windows

To accomplish this:	Use one of these methods
Open an edit window	Choose File | Open to open a file and display it in a window.
Close a window	Press *Ctrl+F4*, choose File | Close from the window Control menu, or double-click the window Control menu button.
Activate a window	Click anywhere in the window you want to activate, select the window name from the Window menu, or press *Ctrl+F6* or *Ctrl+Tab* until the window you want is selected.
Move the active window	Drag the title bar of the window with your mouse, or select Move from the window Control menu and use your cursor keys to move the window; press *Enter*.
Resize the active window	Drag the border of the window you want to change with your mouse, or select Size from the window Control menu and use your cursor keys to resize the window; press *Enter*.
Maximize the active window	Click the Maximize button on the title bar, or select Maximize from the window Control menu.

Table 1.4: Manipulating windows (continued)

To accomplish this:	Use one of these methods
Shrink the active window	Click the Minimize button on the title bar, or select Minimize from the window Control menu.
Restore a window icon	Double-click the window icon, or select it with *Ctrl+F6* or *Ctrl+Tab*, press *Alt+Hyphen* (-) to display the window Control menu, and select Restore.
Restore the active window	Click the restore button, or choose Restore from the window Control menu.
Go to the next window	Click the window you want, press *Ctrl+F6* or *Ctrl+Tab*, or choose Next from the Control menu .

Dialog boxes

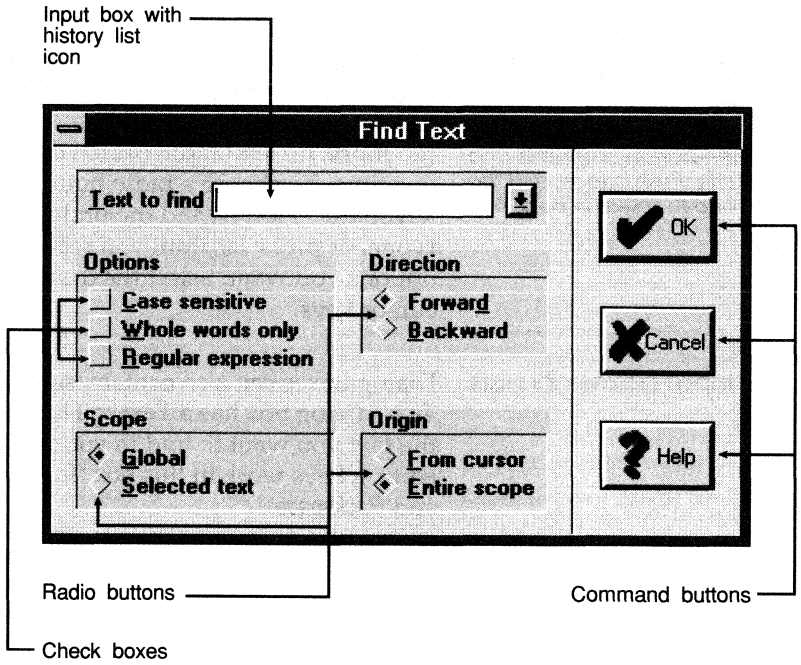
If a menu command has an ellipsis after it (...), the command opens a *dialog box*. A dialog box is a convenient way to view and set multiple options.

Alt+F4 or Esc will close a dialog box, but will not save any settings made in the dialog box.

Each dialog box has its own Control-menu box. Use it to move or close the dialog box. Note that the keyboard shortcut to close a dialog box is *Alt+F4*, which differs from the shortcut to close an edit window (*Ctrl+F4*).

When you're selecting options in dialog boxes, you work with five basic types of onscreen controls: command buttons, check boxes, radio buttons, input boxes, and list boxes. Here's a typical dialog box that illustrates some of these items:

Figure 1.3
A typical dialog box



Command buttons

Esc is a keyboard shortcut for Cancel.

The Find Text dialog box has three command buttons: OK, Cancel, and Help. If you choose OK, Turbo Pascal will search for the text you specified in the input box. If you choose Cancel, nothing changes and no action occurs, but the dialog box is put away. Selecting Help displays a help window for this dialog box.

If you're using a mouse, you can just click the command button you want. Usually OK is the *default button* and will be outlined in black. Using your keyboard, you only need to press *Enter* to choose that button. If you want to choose another command button, press *Tab* until the action you want to select is outlined in black, then press *Enter*.

Check boxes and radio buttons

You can press Shift+Tab to tab back to a previous option.

The Find Text dialog box also has *check boxes* for various options. When you check a check box, a check mark appears in it to show you it's on. An empty box indicates it's off. To check a check box (set it to on), you can either click it or its text, press *Tab* until the check box is outlined and then press *Spacebar*, or, if a letter in the option you want is underlined, you can press *Alt* and type the

underlined letter of the option you want to select. Any number of check boxes can be checked at any time.

Radio buttons act like the old-style station-selection buttons on a car radio. There is always one—and only one—button pushed in at a time. Push one in, and the one that was in pops out.

The Find dialog box also has *radio buttons*, which 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 underlined letter in the option, if there is one, or press *Tab* until you reach the group of radio button options you want. Select the desired radio button with your cursor keys.

Input boxes and lists

Dialog boxes can also contain input boxes. For example, the Find Text dialog box has a Text to Find input box in which you enter the text you want to find in your edit window. Most basic text-editing keys work in this box (for example, arrow keys, *Home*, *End*, and *Backspace*).

If an input box has a drop-down arrow to its right, you can display a *list* of options. With your mouse, click the drop-down arrow, or press *Alt+* ↓ to pull down the *list box* to see all your choices. You can click the option you want or use the arrow keys to highlight your choice and press *Enter*. Press *Alt+* ↑ to put the list box away.

Most of the list boxes in Turbo Pascal are actually *history lists*. For instance, if you use the Find Text dialog box to search for different strings several times, the Text to Find history list will remember the previous strings you entered and display them for you.

You can also simply press ↓ while you are in an input box and your previous entry will appear. Continuing to press ↓ will eventually display all your previous choices. ↑ will take you back through the list.

The status bar

A status bar at the bottom of the Turbo Pascal desktop indicates what is happening in the active edit window. The line and column indicators tell you the location of your cursor. If you edit the text in your window, a *modified* indicator appears. The status bar also tells you if you are in *insert* or *overwrite* mode. Insert mode is Turbo Pascal's default setting. The status bar gives you hints about what a menu command does when the command is

highlighted. Finally, Turbo Pascal displays error messages on the status bar.

Figure 1.4
The status bar

Editing

As a user of Windows, you are probably familiar with editor commands in other Windows products. Turbo Pascal supports all the editing commands common to Windows programs.

For a discussion on editing your text, Turbo style, see Chapter 7.

If you have used Borland language products before, you may prefer to use Borland editing commands. To make your Turbo Pascal for Windows editor a Turbo-style editor, choose the Preference command from the Options menu, find the Command Set group, and select Alternate.

Creating your first program

All these examples are on your distribution diskettes.

Now that you are familiar with the IDE, it's time to write your first program. Begin typing in the following program, pressing *Enter* at the end of each line.

*Don't forget the semicolons, and follow the last **end** with a period.*

```
program MyFirst;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
begin
  Write('Enter two numbers: ');
  Readln(A, B);
  Ratio := A / B;
  Writeln('The ratio is ', Ratio);
end.
```

Use the *Backspace* key to make deletions, and use the arrow keys to move around in the edit window. If you're unfamiliar with editing commands, Chapter 7 discusses all those available.

Analyzing your first program

While you can type in and run this program without ever knowing how it works, we've provided a brief explanation here. The first line you entered gives the program the name *MyFirst*. This is an optional statement, but it's a good practice to include it.

The **uses** clause says that the program uses a unit named *WinCrt*. A *unit* is a library, or collection, of subroutines (procedures and functions) and other declarations. In this case, the *WinCrt* unit opens a window to display the output of your program.

The next three lines declare some *variables*, with the word **var** signaling the start of variable declarations. *A* and *B* are declared to be of type Integer; that is, they can contain whole numbers, such as 52, –421, 0, 32,283, and so on. *Ratio* is declared to be of type Real, which means it can hold fractional numbers such as 423.328 and –0.032, in addition to all integer values.

The rest of the program contains the *statements* to be executed. The word **begin** signals the start of the program. The statements are separated by semicolons and contain instructions to write to the screen (*Write* and *Writeln*), to read from the keyboard (*Readln*), and to perform calculations (*Ratio* := *A* / *B*). The program's execution starts with the first instruction after **begin** and continues until **end.** is encountered.

Saving your first program

Never save a program with a NONAMExx.PAS name.

After entering your first program, it's a good idea to save it to disk. Choose the Save command from the File menu (*Alt+F S* or use your mouse). Since you have not changed the default name of your file (`NONAME00.PAS`), Turbo Pascal will display the Save As dialog box so you can enter a new file name. You're in the input box, so go ahead and type `MYFIRST` (you don't need the .PAS extension; it's assumed). Choose OK.

Compiling your first program

To compile your first program, choose the Compile option on the Compile menu (*Alt+C C* or use your mouse). Turbo Pascal compiles your program, changing it from Pascal (which you can read) to machine code for your computer's microprocessor (which your

PC can execute). You don't see the machine code; it's stored on disk in an .EXE program file.

Like English, Pascal has rules of grammar you must follow. Unlike English, however, Pascal's structure isn't lenient enough to allow for slang or poor syntax—the compiler must always understand what you mean. In Pascal, when you don't use the appropriate words or symbols in a statement or when you organize them incorrectly, you will get a compile-time (syntax) error.

What compile-time errors are you likely to get? Probably the most common error novice Pascal programmers get is

```
Unknown identifier
```

or

```
';' expected
```

Pascal requires that you declare all variables, data types, constants, and subroutines—in short, all identifiers—before using them. If you refer to an undeclared identifier or if you misspell it, you'll get an error. Other common errors are unmatched **begin..end** pairs, assignment of incompatible data types (such as assigning reals to integers), parameter count and type mismatches in procedure and function calls, and so on.

When you start compiling, a Compile Status dialog box appears on your screen, giving you information about the compilation taking place. If no errors occur, the message "Successfully completed" appears in the Status box. Choose OK to put the dialog box away.

If an error occurred during compilation, Turbo Pascal stops, the Compile Status dialog goes away automatically, an error message appears on the status bar, and the line that caused the error is highlighted in your edit window. The next time you click your mouse or press a key, the error message will clear (you can get it back with the Search | Show Last Compile Error). Make the correction, save the updated file, and compile it again.

Running your first program

After you've fixed any typing errors that might have occurred, go to the main menu and choose Run | Run. A new window opens up and the message

```
Enter two numbers:
```

appears in the window. Type in any two integers (whole numbers), with a space between them, and press *Enter*. The following message appears:

```
The ratio is
```

followed by the ratio of the first number to the second. If your program runs without errors, the program window becomes inactive (the word "Inactive" appears on the window title bar) when the program is finished. Close the program window to return to your edit window.

If an error occurs while your program is executing, your program window becomes inactive *before* your program finishes. To find out more information, close the program window and an information box will appear giving you the error number and the memory address where the error occurred. For example,

```
Runtime error 200 at 10BD:0098
```

See Appendix A, "Error messages," in the Programmer's Guide for information on compiler and run-time error messages.

10BD is the segment; 0098 is the offset. Choose OK to put the information box away.

You can now modify your program if you wish. Recompile it and run it again.

Checking the files you created

If you select the File | Open command, you will see the Pascal source file you created in the files list (`MYFIRST.PAS`). If you type `*.*` in the input box and choose OK, you will see all the files in your directory including your executable file (`MYFIRST.EXE`). When you are done examining your files, put away the dialog box by choosing Cancel or pressing *Esc*. (Don't choose OK or you will open a file.)

Stepping up: Your second program

Now you're going to write a second program that builds upon the first. Modify your MYFIRST.PAS program to look like this:

```
program MySecond;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A, B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio:8:2);
  until B = 0;
end.
```

You want to save this as a separate program. To do this, go to the File menu, select Save As, type `MYSECOND.PAS`, and choose OK. Compile the program.

A major change has been made to the program: The statements have been enclosed in the **repeat..until** loop. This causes all the statements between **repeat** and **until** to be executed until the expression following **until** is True. A test is made to see if `B` has a value of zero or not. If `B` has a value of zero, then the loop should exit.

Run your program, try out some values, then type `1 0` and press *Enter*. Your program stops with a run-time error. Close the program window to see the run-time error message, and write down the address of the error. For example, if you see the message

Runtime error 200 at 0001:006A

write down 0001:006A. Choose the Find Error command on the Search menu and the Find Error dialog box will appear. Type the address you just wrote down in the input box and choose OK. You'll notice the compiler recompiling your program. It stops when it reaches the address where the run-time error occurred and it highlights the statement that caused the run-time error.

Debugging your program

If you've programmed before, you can probably guess what this error is and know how to fix it. But let's take this opportunity to show you how to use Turbo Debugger for Windows to find the problem.

Turbo Debugger for Windows allows you to step through your code one line at a time. At the same time, you can watch your variables to see how their values change.

Before you begin debugging, you must be sure the necessary debug information gets into your executable file. Choose the Options | Linker command and check Debug Info in EXE in the Linker Options dialog box. Choose OK.

To start the debugging session, choose the Run | Debugger command. If your program needs to be recompiled, Turbo Pascal will do so. After a brief pause, Turbo Debugger for Windows starts up with your program loaded and a pointer pointing at the first statement (**begin** in this case) in the main body of your program.

Choose the Run | Trace Into command (or press *F7*) to begin executing the program. The debugger just executed the **begin** startup code. The next executable line in this program is the *Write* statement.

Press *F7* again. Your screen blinks momentarily, then shows your program with the pointer on the second statement (*Readln*). What's happening here is that Turbo Pascal switches to the User screen (where your program is executed and its output is displayed), executes your first statement (a *Write* statement), then goes back to the Turbo Debugger screen.

Press *F7* again. This time, the User screen comes up and stays there. That's because a *Readln* statement is waiting for you to enter two numbers. Type two integer numbers, separated by a space; be sure the second number isn't a zero. Now press *Enter*. You're back at the Turbo Debugger screen, with the pointer pointing at the assignment statement (*Ratio := A / B*).

Press *F7* and execute the assignment statements. Now the pointer is on the *Writeln* statement. Press *F7* to execute it. To see your program screen, choose Turbo Debugger's Window | User screen command from the menus or press the shortcut key *Alt+F5*. When you are ready to return to Turbo Debugger, press any key.

The pointer is now on the **until** clause. Press *F7* one more time, and you're back at the top of the **repeat** loop.

Instead of racing through one program statement after another, Turbo Debugger for Windows lets you step through your code one line at a time. This is a powerful tool, and we go into a more detailed discussion of debugging in the *Turbo Debugger for*

Windows User's Guide. First, we'll give you a quick taste of debugging by tracking down that run-time error.

Let's take a look at the values of the variables you've declared. Press View | Watches to go to the Watches window at the bottom of the Turbo Debugger screen. Type `A` in the Watch window. When you begin typing, the Expression to Watch dialog box appears. Choose OK. This puts *A* in the Watch window, along with its current value. Now repeat the same steps to add *B* and *Ratio* to the Watch window. Finally, use it to add the expression *A / B* to the Watch window.

Choose Run | Trace Into (or press *F7*) to step through your program. This time, when you have to enter two numbers, enter `0` for the second number. When you press *Enter* and return to the Turbo Debugger screen, look at the expression A / B in the Watch window. Instead of having a value after it, it has this notation:

```
+INF
```

This stands for "positive infinity", which means dividing by zero is undefined. Note, though, that having this expression in your Watch window doesn't cause the program to stop with an error. Instead, the error is reported to you and the debugger does not perform the division in the Watch window.

Now press *F7* again, and assign *A/B* to *Ratio*. At this point, your program does halt and your User screen window becomes inactive. Close the User screen window and Turbo Pascal will display a run-time error message in an information box. Choose OK to put the information box away. You will be back in Turbo Debugger in an information box that displays this message:

```
Terminated, exit code 0
```

Choose OK to put the information box away and exit Turbo Debugger with File | Quit (*Alt+X*).

Fixing your second program

Now you probably have a good idea of what's wrong with your program: If you enter a value of zero for the second number (`B`), the program halts with a run-time error.

How do you fix it? If `B` has a value of zero, don't divide `B` into `A`. Edit your program so that it looks like this:

```
program MySecond;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A, B);
    if B = 0 then
      Writeln('The ratio is undefined')
    else
    begin
      Ratio := A / B;
      Writeln('The ratio is ', Ratio:8:2);
    end;
  until B = 0;
end.
```

Be sure to save your changes. Now run your program (either by itself, or using the debugger). If you use the debugger, note how the values in the Watch window change as you step through the program. When you're ready to stop, enter `0` for *B*. The program will end after displaying the message "The ratio is undefined." Close the inactive program window.

Now you have an idea just how powerful Turbo Debugger for Windows is. You can step through your program line by line, you can display the value of your program's variables and expressions, and you can watch the values change as your program runs. Read more about debugging your Windows programs with Turbo Debugger in the the *Turbo Debugger for Windows User's Guide*.

C H A P T E R

2

Programming in Turbo Pascal

The Pascal language was designed by Niklaus Wirth in the early 1970s to teach programming. Because of that, it's particularly well-suited as a first programming language. And if you've already programmed in other languages, you'll find it easy to pick up Pascal.

To get you started on the road to Pascal programming, this chapter will teach you the basic elements of the Pascal language and show you how to use them in your programs.

Writing Windows applications with text-oriented output can be complicated, but with Turbo Pascal, you can develop simple, text-based programs using the *WinCrt* unit. You'll learn about the *WinCrt* unit in Chapter 3, "Turbo Pascal units."

All example programs in this chapter use the *WinCrt* unit to display their output. As you become more advanced, you'll write most of your programs with ObjectWindows and you won't need the *WinCrt* unit. As we discuss the basics of the Pascal language, however, you'll find concepts easier to understand if you needn't worry about the special requirements of Windows. Therefore, all the information about programming in Pascal presented in this chapter refers to text-based programs developed using the *WinCrt* unit.

Before you work through this chapter, you might want to read Chapter 6, "The IDE reference," and Chapter 7, "The editor from A to Z," to learn about the environment and text editor in Turbo

Pascal. If you haven't already installed Turbo Pascal as described in the introduction, you should do so now.

The elements of programming

Most programs are designed to solve a problem by manipulating information or data. What you as the programmer have to do is

- get the information into the program—*input*.
- have a place to keep it—*data*.
- give the right instructions to manipulate the data—*operations*.
- be able to get the data back out of the program to the user (you, usually)—*output*.

You can organize your instructions so that

- some are executed only when a specific condition (or set of conditions) is True—*conditional execution*.
- others are repeated a number of times—*loops*.
- others are broken off into chunks that can be executed at different locations in your program—*subroutines*.

These are the seven basic elements of programming: *input*, *data*, *operations*, *output*, *conditional execution*, *loops*, and *subroutines*. This list is not comprehensive, but it does describe those elements that programs (and programming languages) usually have in common.

Many programming languages, including Pascal, have additional features. When you want to learn a new language quickly, find out how that language implements these seven elements, then build from there. Here's a brief description of each element

Input
This means reading values in from the keyboard, from a disk, or from an I/O port.

Data
These are constants, variables, and structures that contain numbers (integer and real), text (characters and strings), or addresses (of variables and structures).

Operations
These assign one value to another, combine values (add, divide, and so forth), and compare values (equal, not equal, and so on).

Output
This means writing information to the screen, to a disk, or to an I/O port.

Conditional execution
This refers to executing a set of instructions if a specified condition is True (and skipping them or executing a different set if it is False) or if a data item has a specified value or range of values.

Loops
These execute a set of instructions some fixed number of times, while some condition is True or until some condition is True.

Subroutines
These are separately named sets of instructions that can be executed anywhere in the program just by referencing the name.

Now we'll take a look at how to use these elements in Turbo Pascal.

Data types

When you write a program, you're working with information that generally falls into one of five basic types: integers, real numbers, characters and strings, Boolean, and pointers.

Integers are the whole numbers you learned to count with (1, 5, –21, and 752, for example).

Real numbers have fractional portions (3.14159) and exponents (2.579×10^{24}). These are also sometimes known as *floating-point numbers.*

Characters are letters of the alphabet, symbols, and the numbers 0-9. They can be used individually (*a, Z,* !, 3) or combined into character strings ('This is only a test.').

Boolean expressions have one of two possible values: True or False. They are used in conditional expressions, which we'll discuss later.

Pointers hold the addresses of locations in the computer's memory, which in turn hold information.

Integer data types

Standard Pascal defines the data type integer as consisting of the values ranging from *–MaxInt* through 0 to *MaxInt*, where *MaxInt* is the largest possible integer value allowed by the compiler you're using. Turbo Pascal supports type integer, defines *MaxInt* as equal to 32,767, and allows the value –32,768 as well. A variable of type integer occupies 2 bytes.

Turbo Pascal also defines a long integer constant, *MaxLongInt*, with a value of 2,147,483,647.

Turbo Pascal also supports four other integer data types, each of which has a different range of values. Table 2.1 shows all five integer types.

Table 2.1
Integer data types

Type	Range	Size in Bytes
Byte	0..255	1
Shortint	–128..127	1
Integer	–32768..32767	2
Word	0..65535	2
Longint	–2147483648..2147483647	4

A final note: Turbo Pascal allows you to use hexadecimal (base 16) integer values. To specify a constant value as hexadecimal, place a dollar sign (**$**) in front of it; for example, $27 = 39 decimal.

Real data types

Standard Pascal defines the data type Real as representing floating-point values consisting of a significand (fractional portion) multiplied by an exponent (power of 10). The number of digits (known as *significant digits*) in the significand and the range of values of the exponent are compiler-dependent. Turbo Pascal defines the type real as being 6 bytes in size, with 11 significant digits and an exponent range of 10^{-38} to 10^{38}.

Turbo Pascal also supports the IEEE Standard 754 for binary floating-point arithmetic. This includes the data types single, double, extended, and comp. Single uses 4 bytes, with 7 significant digits and an exponent range of 10^{-45} to 10^{38}; double uses 8 bytes, with 15 significant digits and an exponent range of 10^{-324} to 10^{308}; and extended uses 10 bytes, with 19 significant digits and an exponent range of 10^{-4951} to 10^{4931}.

If you have an 80x87 math coprocessor and enable the numeric support compiler directive or environment option ({**$N+**}), Turbo Pascal generates the proper coprocessor instructions to support these types and to perform all floating-point operations on the coprocessor. If you don't have an 80x87 coprocessor, Turbo Pascal will automatically use Windows software routines to *emulate* the coprocessor.

Table 2.2
Real data types

Type	Range	Significant Digits	Size in Bytes
real	2.9 × 10E-39 .. 1.7 × 10E38	11-12	6
single	1.5 × 10E-45 .. 3.4 × 10E38	7- 8	4
double	5.0 × 10E-324 .. 1.7 × 10E308	15-16	8
extended	3.4 × 10E-4932 .. 1.1 × 10E4932	19-20	10
comp*	–2E+63+1 .. 2E+63-1	19-20	8

* comp only holds integer values.

Get into the Turbo Pascal editor and enter the following program:

```
program DoRatio;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
begin
  Write('Enter two numbers:  ');
  Readln(A, B);
  Ratio := A div B;
  Writeln('The ratio is ', Ratio);
end.
```

Save this as DORATIO.PAS by selecting File | Save As from the main menu. Then press *Ctrl+F9* to compile and run the program. Enter two values (such as 10 and 3) and note the result (3.0000000000E+00). You probably expected an answer of 3.3333333333E+00, but instead you received a 3. That's because you used the **div** operator, which performs integer division. Close the WinCrt window and go back and change the **div** statement to read as follows:

```
Ratio := A / B;
```

Save the code (File | Save), then compile and run. The result is now 3.3333333333, as you expected. Using the division operator (*/*) gives you the most precise result—a real number.

Character and string data types

You've learned how to store numbers in Pascal, now how about characters and strings? Pascal offers a predefined data type Char that is 1 byte in size and holds exactly one character. Character constants are represented by surrounding the character with single quotes (for example, 'A', 'e', '?', '2'). Note that '2' means the *character* 2, while 2 means the integer 2 (and 2.0 means the *real number* 2).

Here's a modification of DORATIO that allows you to repeat it several times (this also uses a **repeat..until** loop, which we'll discuss a little later):

```
program DoRatio;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
  Ans: Char;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A, B);
    Ratio := A / B;
    Writeln('The ratio is ', Ratio);
    Write('Do it again? (Y/N)  ');
    Readln(Ans);
  until UpCase(Ans) = 'N';
end.
```

After calculating the ratio once, the program writes the message

```
Do it again? (Y/N)
```

and waits for you to type in a single character and then press *Enter*. If you type in a lowercase *n* or an uppercase *N*, the **until** condition is met and the loop ends; otherwise, the program goes back to the **repeat** statement and starts over again.

Appendix B in the Programmer's Guide lists the ASCII codes for all characters.

Note that *n* is *not* the same as *N*. This is because they have different ASCII code values. Characters are represented by the ASCII code: Each character has its own 8-bit number (characters take up 1 byte, remember).

Turbo Pascal gives you two additional ways of representing character constants: with a caret (^) or a number symbol (#). First,

the characters with codes 0 through 31 are known as *control characters* (because historically they were used to control teletype operations). They are referred to by their abbreviations (CR for carriage return, LF for linefeed, ESC for escape, and so on) or by the word "Ctrl" followed by a corresponding letter (meaning the letter produced by adding 64 to the control code). For example, the control character with ASCII code 7 is known as BEL or *Ctrl-G*. Turbo Pascal lets you represent these characters using the caret (^), followed by the corresponding letter (or character). Thus, ^G is a legal representation in your program of *Ctrl-G*, and you could write statements such as *Writeln(^G)*, causing your computer to beep at you. This method, however, only works for the control characters.

You can also represent *any* character using the number symbol (#), followed by the character's ASCII code. Thus, #7 is the same as ^G, #65 is the same as 'A', and #233 represents one of the special IBM PC graphics characters.

Individual characters are nice, but what about strings of characters? After all, that's how you will most often use them. Standard Pascal does not support a separate string data type, but Turbo Pascal does. Take a look at this program:

```
program Hello;
uses WinCrt;
var
  Name: string[30];
begin
  Write('What is your name?  ');
  Readln(Name);
  Writeln('Hello, ', Name);
end.
```

This declares the variable *Name* to be of type string, with space set aside to hold 30 characters. One more byte is set aside internally by Turbo Pascal to hold the current length of the string. That way, at the prompt, you can enter any name for the *Writeln* statement to print out. However, if you enter a name that is more than 30 characters, the *Writeln* statement prints only the first 30 characters and ignores the rest.

When you declare a string variable, you can specify how many characters (up to 255) it can hold. Or you can declare a variable (or parameter) to be of type String with no length mentioned, in which case the default size of 255 characters is assumed.

Turbo Pascal offers a number of predefined procedures and functions to use with strings; they can be found in Chapter 24 of the *Programmer's Guide*. When writing programs for Windows, you may need to use special string types and functions; see Chapter 13, "The Strings unit" for more information.

Boolean data types

Turbo Pascal's predefined data types, Boolean, Wordbool, and Longbool, have two possible values: True and False. You can declare a variable to be of one of these types, then assign the variable either a True or False value or (more importantly) an expression that resolves to one of those two values.

A *Boolean expression* is simply an expression that is either True or False. It is made up of relational expressions, Boolean operators, Boolean variables, and/or other Boolean expressions. For example, the following **while** statement contains a Boolean expression:

```
while (Index <= Limit) and not Done do ...]
```

The Boolean expression consists of everything between the keywords **while** and **do**, and presumes that *Done* is a variable (or possibly a function) of type Boolean, Wordbool, or Longbool.

Pointer data type

All the data types we've discussed until now hold just that—data. A pointer holds a different type of information—an address. A pointer is a variable that contains the address in memory (RAM) where some data is stored, rather than the data itself. In other words, it points to the data, like an address book or an index.

A pointer is usually (but not necessarily) specific to some other data type. Consider the following declarations:

```
type
  TBuffer  = string[255];
  PBuffer  = ^Buffer;
var
  Buf1: TBuffer;
  Buf2: PBuf;
```

The data type *TBuffer* is now just another name for **string**[255], while the type *PBuffer* defines a pointer to a *TBuffer*. The variable *Buf1* is of type *TBuffer*; it takes up 256 bytes of memory. The

variable *Buf2* is of type *PBuffer*; it contains a 32-bit address and takes up only 4 bytes of memory.

Where does *Buf2* point? Nowhere, currently. Before you can use *PBuffer*, you need to set aside (allocate) some memory and store its address in *Buf2*. You do that using the *New* procedure:

```
New(Buf2);
```

Since *Buf2* points to the type *TBuffer*, this statement creates a 256-byte buffer somewhere in memory, then puts its address into *Buf2*.

How do you use the data pointed to by *Buf2*? Via the indirection operator **^**. For example, suppose you want to store a string in both *Buf1* and the buffer pointed to by *Buf2*. Here's what the statements would look like:

```
Buf1 := 'This string gets stored in Buf1.'
Buf2^ := 'This string gets stored where Buf2 points.'
```

Note the difference between *Buf2* and *Buf2^*. *Buf2* refers to a 4-byte pointer variable; *Buf2^* refers to a 256-byte string variable whose address is stored in *Buf2*.

How do you free up the memory pointed to by *Buf2*? By using the *Dispose* procedure. *Dispose* makes the memory available for other uses. After you use *Dispose* on a pointer, it's good practice to assign the (predefined) value **nil** to that pointer. That lets you know that the pointer no longer points to anything:

```
Dispose(Buf2);
Buf2 := nil;
```

Note that you assign **nil** to *Buf2*, not to *Buf2^*.

Identifiers

Up until now, we've given names to variables without worrying about what restrictions there might be. Let's talk about those restrictions now.

The names you give to constants, data types, variables, and functions are known as *identifiers*. Some of the identifiers used so far include

integer, real, string	Predefined data types
Hello,DoRatio	Programs
Name, A, B, Ratio	User-defined variables
Write,Writeln,Readln	Predeclared procedures

Turbo Pascal has a few rules about identifiers; here's a quick summary:

- All identifiers must start with a letter or underscore (*a*...*z*, *A*...*Z*, or _). The rest of an identifier can consist of letters, underscores, and/or digits (0...9); no other characters are allowed.
- Identifiers are *case-insensitive*, which means that lowercase letters (*a*...*z*) are considered the same as uppercase letters (*A*...*Z*). For example, the identifiers *indx*, *Indx*, and *INDX* are identical.
- Identifiers can be of any length, but only the first 63 characters are significant.

Operators

Once you get your data into the program (and into your variables), you'll probably want to manipulate it somehow, using the operators available to you. There are eight operator types: assignment, arithmetic, bitwise, relational, logical, address, set, and string.

Most Pascal operators are *binary*, taking two operands; the rest are *unary*, taking only one operand. Binary operators use the usual algebraic form, for example, *a* + *b*. A unary operator always precedes its operand, for example, –*b*.

In more complex expressions, rules of precedence clarify the order in which operations are performed (see Table 2.3).

Table 2.3
Operator precedence

Operators	Precedence	Categories
@, not, ^	First (high)	Unary operators
*, /, div, mod, and, shl, shr	Second	Multiplying operators
+,-, or, xor	Third	Adding operators
=, <>, <, >, <=, >=, in	Fourth (low)	Relational operators

Operations with equal precedence are normally performed from left to right, although the compiler may at times rearrange the operands to generate optimum code.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators.

Assignment operators

The most basic operation is *assignment* (that is, assigning a value to a variable), as in *Ratio := A / B*. In Pascal, the assignment symbol is a colon followed by an equal sign (**:=**). In the example given, the value of *A / B* on the right of the assignment symbol is assigned to the variable *Ratio* on the left.

Arithmetic operators

Pascal supports the usual set of binary arithmetic operators—they work with type integer and real values:

- Multiplication (*)
- Integer division (**div**)
- Real division (*/*)
- Modulus (**mod**)
- Addition (**+**)
- Subtraction (–)

Also, Turbo Pascal supports both *unary minus* (*a + (-b)*), which performs a *two's complement* evaluation, and *unary plus* (*a + (+b)*), which does nothing at all but is there for completeness.

Bitwise operators

For bit-level operations, Pascal has the following operators:

- **shl** (shift left): Shifts the bits left the indicated number of bits, filling at the right with 0's.
- **shr** (shift right): Shifts the bits right the indicated number of bits, filling at the left with 0's.
- **and**: Performs a logical **and** on each corresponding pair of bits, returning 1 if both bits are 1, and 0 otherwise.
- **or**: Performs a logical **or** on each corresponding pair of bits, returning 0 if both bits are 0, and 1 otherwise.
- **xor**: Performs a logical, exclusive **or** on each corresponding pair of bits, returning 1 if the two bits are different from one another, and 0 otherwise.

- **not**: Performs a logical complement on each bit, changing each 0 to a 1, and vice versa.

These allow you to perform very low-level operations on integer values.

Relational operators

Relational operators allow you to compare two values, yielding a Boolean result of True or False. Here are the relational operators in Pascal:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to
in	is a member of

Why would you want to know if something were True or False? Enter the following program:

```
program TestGreater;
uses WinCrt;
var
  A, B: Integer;
  Test: Boolean;
begin
  Write('Enter two numbers:  ');
  Readln(A, B);
  Test := A > B;
  Writeln('A is greater than B ', Test);
end.
```

This will print True if *A* is greater than *B* or False if *A* is less than or equal to *B*.

Logical operators

There are four logical operators—**and**, **xor**, **or**, and **not**—that are similar to but not identical with the bitwise operators. These logical operators work with logical values (True and False), allowing you to combine relational expressions, Boolean variables, and Boolean expressions.

They differ from the corresponding bitwise operators in this manner:

- Logical operators always produce a result of either True or False (a Boolean value), while the bitwise operators do bit-by-bit operations on type integer values.
- You cannot combine Boolean and integer-type expressions with these operators; in other words, the expression `Flag` **and** `Indx` is illegal if *Flag* is of type Boolean, and *Indx* is of type Integer (or vice versa).
- The logical operators **and** and **or** will short-circuit by default; **xor** and **not** will not. Suppose you have the expression `exp1` **and** `exp2`. If *exp*1 is False, then the entire expression is False, so *exp*2 will never be evaluated. Likewise, given the expression `exp1` **or** `exp2`, *exp*2 will never be evaluated if *exp*1 is True. You can force full Boolean expression using the {**$B+**} compiler directive or the Complete Boolean Evaluation option (Options | Compiler).

Address operators

Pascal supports two special address operators: the *address-of* operator (@) and the *indirection* operator (^).

The @ operator returns the address of a given variable; if *Sum* is a variable of type Integer, then @*Sum* is the address (memory location) of that variable. Likewise, if *ChrPtr* is a pointer to type Char, then *ChrPtr*^ is the character to which *ChrPtr* points.

Set operators

Set operators perform according to the rules of set logic. The set operators and operations include

+	union
–	difference
*	intersection

String operator

The only string operation is the + operator, which is used to concatenate two null-terminated strings.

Output

It may seem funny to talk about output before input, but a program that doesn't output information isn't of much use. That

output usually takes the form of information written to the screen (words and pictures), to a storage device (floppy or hard disk), or to an I/O port (serial or printer ports).

The Writeln procedure

You've already used the most common output function in Pascal, the *Writeln* routine. The purpose of *Writeln* is to write information to the screen. Its format is both simple and flexible:

```
Writeln(item, item,...);
```

Each *item* is something you want to print to the screen. It can be a literal value, such as an integer or a real number (3, 42, –1732.3), a character ('a', 'Z'), a string ('Hello, world'), or a Boolean, WordBool, or LongBool value (True). It can also be a named constant, a variable, a dereferenced pointer, or a function call, as long as it yields a value that is of type Integer, Real, Char, String, or Boolean. All the items are printed on one line, in the order given. The cursor is then moved to the start of the next line. If you want to leave the cursor after the last item on the same line, then use the statement

```
Write(item, item,...);
```

When the items in a *Writeln* statement are printed, blanks are *not* automatically inserted; if you want spaces between items, you'll have to put them there yourself, like this:

```
Writeln(item, ' ', item,' ',...);
```

For example, the following statements produce the output shown at the right:

```
A := 1; B := 2; C := 3;
Name := 'Frank';
Writeln(A,B,C);                    123
Writeln(A,' ', B, ' ', C);           1 2 3
Writeln('Hi', Name);                HiFrank
Writeln('Hi, ', Name,'.');          Hi, Frank.
```

You can also use *field-width specifiers* to define a field width for a given item. The format for this is

```
Writeln(item:width,...);
```

where *width* is an integer expression (literal, constant, variable, function call, or combination thereof) specifying the total width of

the field in which *item* is written. For example, consider the following code and resulting output:

```
A := 10; B := 2; C := 100;
Writeln(A,B,C);                    102100
Writeln(A:2,B:2,C:2);              10 2100
Writeln(A:3,B:3,C:3);               10  2100
Writeln(A,B:2,C:4);                10 2 100
```

Note that the item is padded with leading blanks on the left to make it equal to the field width. The actual value is right-justified.

What if the field width is less than what is needed? In the second *Writeln* statement in the previous code, *C* has a field width of 2 but has a value of 100 and needs a width of 3. As you can see by the output, Pascal simply expands the width to the minimum size needed.

This method works for all allowable items: integers, reals, characters, strings, and Booleans. However, real numbers printed with the field-width specifier (or with none at all) come out in exponential form:

```
X := 421.53;
Writeln(X);                        4.2153000000E+02
Writeln(X:8);                      4.2E+02
```

Because of this, Pascal allows you to append a second field-width specifier: *item:width:digits*. This second value forces the real number to be printed out in fixed-point format and tells how many digits to place after the decimal point:

```
X := 421.53;
Writeln(X:6:2);                    421.53
Writeln(X:8:2);                      421.53
Writeln(X:8:4);                    421.5300
```

Input

Standard Pascal has two basic input functions that are used to read from data from the keyboard: *Read* and *Readln*. The general syntax is

```
Read(item, item,...);
```

or

```
Readln(item, item,...);
```

Each *item* is a variable of any integer, real, Char, or string type. Numbers being input must be separated from other values by spaces or by pressing *Enter*.

Conditional statements

There are times when you want to execute some portion of your program when a given condition is True or False, or when a particular value of a given expression is reached. Let's look at how to do this in Pascal.

The if statement

Look again at the **if** statement in the previous examples; note that it can take the following generic format:

```
if expr
  then statement1
  else statement2
```

expr is any Boolean expression (resolving to True or False), and *statement1* and *statement2* are legal Pascal statements. If *expr* is True, then *statement1* is executed; otherwise, *statement2* is executed.

Two important points about **if/then/else** statements:

1. **else** *statement2* is optional; in other words, this is a valid **if** statement:

   ```
   if expr
     then statement1
   ```

   In this case, *statement1* is executed only if *expr* is True. If *expr* is False, then *statement1* is skipped, and the program continues.
2. What if you want to execute more than one statement if a particular expression is True or False? You use a compound statement. A *compound statement* consists of the keyword **begin**, some number of statements separated by semicolons (;), and the keyword **end**.

The ratio example uses a single statement for the **if** clause,

```
if B = 0 then
  Writeln('The ratio is undefined.')
```

and a compound statement for the **else** clause

```
else
begin
  Ratio = A / B;
  Writeln('The ratio is ', Ratio:8:2);
end;
```

You might also notice that the body of each program you've written is simply a compound statement followed by a period.

The case statement

This statement gives your program the power to choose from more than two alternatives without having to specify lots of **if** statements.

The **case** statement consists of an expression (the selector) and a list of statements, each preceded by a **case** label of the same type as the selector. It specifies that the one statement be executed whose **case** label is equal to the current value of the selector. If none of the **case** labels contain the value of the selector, then either no statement is executed or, optionally, the statements following the reserved word **else** are executed.

*Using **else** as part of the **case** statement is an extension to standard Pascal.*

A **case** label consists of any number of constants or subranges, separated by commas and followed by a colon; for example,

```
case BirdSight of
  'C', 'c': Curlews := Curlews + 1;
  'H', 'h': Herons := Herons + 1;
  'E', 'e': Egrets := Egrets + 1;
  'T', 't': Terns  := Terns + 1;
end;  { case }
```

A subrange is written as two constants separated by the subrange delimiter '..'. The constant type must match the selector type. The statement that follows the **case** label is executed if the selector's value equals one of the constants or if it lies within one of the subranges.

Loops

Just as there are statements (or groups of statements) that you want to execute conditionally, there are other statements that you

may want to execute repeatedly. This kind of construct is known as a *loop*.

There are three basic kinds of loops: the **while** loop, the **repeat..until** loop, and the **for** loop. We'll cover them in that order.

The while loop

You can use the **while** loop to test for something at the beginning of your loop. Enter the following program:

```
program Hello;
uses WinCrt;
var
  Count: Integer;
begin
  Count := 1;
  while Count <= 10 do
  begin
    Writeln('Hello and goodbye!');
    Inc(Count);
  end;
  Writeln('This is the end!');
end.
```

The first thing that happens when you run this program is that *Count* is set equal to 1, then you enter the **while** loop. This tests to see if *Count* is less than or equal to 10. At this point *Count* is less than 10, so the loop's body (**begin..end**) is executed. This prints the message `Hello and goodbye!` to the screen, then increments *Count* by 1. *Count* is again tested, and the loop's body is executed once more. This continues as long as *Count* is less than or equal to 10 when it is tested. Once *Count* reaches 11, the loop stops, and the string `This is the end!` is printed on the screen.

The format of the **while** statement is

```
while expr do statement
```

where *expr* is a Boolean expression, and *statement* is either a single or a compound statement.

The **while** loop evaluates *expr*. If it's True, then *statement* is executed, and *expr* is evaluated again. If *expr* is False, the **while** loop is finished and the program continues.

The repeat..until loop

The second loop is the **repeat..until** loop, which we've seen in the program DORATIO.PAS:

```
program DoRatio;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
  Ans: Char;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A, B);
    Ratio := A / B;
    Writeln('The ratio is ', Ratio);
    Write('Do it again? (Y/N)  ');
    Readln(Ans);
  until UpCase(Ans) = 'N';
end.
```

As described before, this program repeats until you answer *n* or *N* to the question `Do it again? (Y/N)`. In other words, everything between **repeat** and **until** is repeated until the expression following **until** is True.

Here's the generic format for the **repeat..until** loop:

```
repeat
  statement;
  statement;
  ...
  statement;
until expr
```

There are three major differences between the **while** loop and the **repeat** loop. First, the statements in the **repeat** loop always execute at least once, because the test on *expr* is not made until after the **repeat** occurs. By contrast, the **while** loop will skip over its body if the expression is initially False.

Next, the **repeat** loop executes *until* the expression is True, where the **while** loop executes *while* the expression is True. This means that care must be taken in translating from one type of loop to the other. For example, here's the HELLO program rewritten using a **repeat** loop:

```
program Hello;
uses WinCrt;
var
  Count: Integer;
begin
  Count := 1;
  repeat
    Writeln('Hello and goodbye!');
    Inc(Count);
  until Count > 10;
  Writeln('This is the end!');
end.
```

Note that the test is now *Count > 10*, whereas for the **while** loop it was *Count <= 10*.

Finally, the **repeat** loop can hold multiple statements without using a compound statement. Notice that you didn't have to use **begin..end** in the preceding program, but you did for the earlier version using a **while** loop.

Again, be careful to note that the **repeat** loop will always execute at least once. A **while** loop may or may not execute, depending on the value of the expression.

The for loop

The **for** loop is the one found in most major programming languages, including Pascal. However, the Pascal version is simultaneously limited and powerful.

Basically, the **for** loop executes a set of statements some fixed number of times while a variable (known as the *index variable*) steps through a range of values. To see how this works, modify the earlier HELLO program to read as follows:

```
program Hello;
uses WinCrt;
var
  Count: Integer;
begin
  for Count := 1 to 10 do
  Writeln('Hello and goodbye!');
  Writeln('This is the end!');
end.
```

When you run this program, you can see that the loop works the same as the **while** and **repeat** loops already shown and, in fact, is

precisely equivalent to the **while** loop. Here's the generic format of the **for** loop statement:

```
for index := expr1 to expr2 do statement
```

index is a variable of some scalar type (any integer type, char, Boolean, any enumerated type), *expr1* and *expr2* are expressions of some type compatible with *index*, and *statement* is a single or compound statement. *Index* is incremented by one after each time through the loop.

You can also decrement the index variable instead of incrementing it by replacing the keyword **to** with the keyword **downto**.

The **for** loop is equivalent to the following code:

```
index := expr1;
while index <= expr2 do
begin
  statement;
  Inc(index);
end;
```

The main drawback of the **for** loop is that it only allows you to increment or decrement by one. Its main advantages are conciseness and the ability to use char and enumerated types in the range of values.

Procedures and functions

You've learned how to execute code conditionally and iteratively. Now, what if you want to perform the same set of instructions on different sets of data or at different locations in your program? Well, you simply put those statements into a *subroutine*, which you can then call as needed.

In Pascal, there are two types of subroutines: *procedures* and *functions*. The main difference between the two is that a function returns a value and can be used in expressions, like this:

```
X := Sin(A);
```

while a procedure is called to perform one or more tasks:

```
Writeln('This is a test');
```

However, before you learn any more about procedures and functions, you need to understand Pascal program structure.

Program structure

In standard Pascal, programs adhere to a rigid format:

```
program ProgName;
label
  labels;
const
  constant declarations;
type
  data type definitions;
var
  variable declarations;
procedures and functions;
begin
  main body of program
end.
```

You do not have to have all five declaration sections—**label**, **const**, **type**, **var**, and **procedures** and **functions**—in every program. But in standard Pascal, if they do appear, they must be in that order, and each section can appear only once. The declaration section is followed by any procedures and functions you might have, and then by the main body of the program, consisting of some number of statements.

Turbo Pascal gives you tremendous flexibility in your program structure. All it requires is that your program statement (if you have one) be first and that your main program body be last. Between those two, you can have as many declaration sections as you want, in any order you want, with procedures and functions freely mixed in. But identifiers must be defined before they are used, or else a compile-time error will occur.

Procedure and function structure

As mentioned earlier, procedures and functions—known collectively as *subprograms*—appear anywhere before the main body of the program. Procedures use this format:

```
procedure ProcName(parameters);
label
  labels;
const
```

```
  constant declarations;
type
  data type definitions;
var
  variable declarations;
procedures and functions;
begin
  main body of procedure;
end;
```

Functions look just like procedures except that a function declaration starts with a **function** header and ends with a data type for the return value of the function:

```
function FuncName(parameters): data type;
```

As you can see, there are only two differences between this and regular program structure: Procedures or functions start with a **procedure** or **function** header instead of a **program** header, and they end with a semicolon instead of a period. A procedure or function can have its own constants, data types, and variables, and even its own procedures and functions. All these items can only be used with the procedure or function in which they are declared.

Sample program

Here's a version of the DORATIO program that uses a procedure to get the two values, then uses a function to calculate the ratio:

```
program DoRatio;
uses WinCrt;
var
  A, B: Integer;
  Ratio: Real;
procedure GetData(var X, Y: Integer);
begin
  Write('Enter two numbers:  ');
  Readln(X, Y);
end;

function GetRatio(I, J: Integer): Real;
begin
  GetRatio := I / J;
end;

begin
  GetData(A, B);
  Ratio := GetRatio(A, B);
```

```
    Writeln('The ratio is ', Ratio);
end.
```

This isn't exactly an improvement on the original program, being both larger and slower, but it does illustrate how procedures and functions work.

When you compile and run this program, execution starts with the first statement in the main body of the program: `GetData(A,B)`. This type of statement is known as a *procedure call.* Your program handles this call by executing the statements in *GetData*, replacing *X* and *Y* (known as *formal parameters*) with *A* and *B* (known as *actual parameters*). The keyword **var** in front of *X* and *Y* in *GetData*'s procedure statement says that the actual parameters must be variables and that the variable values can be changed and passed back to the caller. So you can't pass literals, constants, expressions, and so on to *GetData*. Once *GetData* is finished, execution returns to the main body of the program and continues with the statement following the call to *GetData*.

That next statement is a function call to *GetRatio*. Note that there are some key differences here. First, *GetRatio* returns a value, which must then be used somehow; in this case, it's assigned to *Ratio*. Second, a value is assigned to *GetRatio* in its main body; this is how a function determines what value to return. Third, there is no **var** keyword in front of the formal parameters *I* and *J*. This means that the actual parameters could be any two integer expressions, such as *Ratio := GetRatio(A + B, 300);* and that even if you change the values of the formal parameters in the **function** body, the new values will not be passed back to the caller. This, by the way, is *not* a distinction between procedures and functions; you can use both types of parameters with either type of subprogram.

Program comments

Sometimes you want to insert notes into your program to remind yourself (or inform someone else) of what certain variables mean, what certain functions or statements do, and so on. These notes are known as *comments*. Pascal, like most other programming languages, lets you put as many comments as you want into your program.

You start a comment with the left curly brace ({), which signals to the compiler to ignore everything until after it sees the right curly brace (}).

Comments can even extend across multiple lines, like this:

```
{ This is a long
  comment, extending
  over several lines. }
```

Pascal also allows an alternative form of comment, beginning with a left parenthesis and an asterisk, **(***, and ending with an asterisk and a right parenthesis, ***)**. This allows for a limited form of comment nesting, because a comment beginning with **(*** ignores all curly braces, and vice versa.

C H A P T E R

3

Turbo Pascal units

This chapter explains what a unit is, how you use it, what predefined units are available, how to go about writing your own units, and how to compile them.

What is a unit?

Turbo Pascal gives you access to a large number of predefined constants, data types, variables, procedures, and functions. Some are specific to Turbo Pascal; others are specific to programming Windows applications. There are dozens of them, but you seldom use them all in a given program. Because of this, they are split into related groups called *units*. You can then use only the units your program needs.

A *unit* is a collection of constants, data types, variables, procedures, and functions. Each unit is almost like a separate Pascal program: It can have a main body that is called before your program starts and does whatever initialization is necessary. In short, a unit is a library of declarations you can pull into your program that allows your program to be split up and separately compiled.

All the declarations within a unit are usually related to one another. For example, the *Strings* unit contains all the declarations for null-terminated string-handling routines.

Turbo Pascal provides six standard units for your use: *System*, *WinCrt*, *Strings*, *WinDos*, *WinTypes*, and *WinProcs*. They provide support for your Turbo Pascal programs and are all stored in TPW.TPL. Some of these are explained more fully in Chapters 11 through 14 of the *Programmer's Guide*, but we'll look at each one here and explain its general function.

A unit's structure

A unit provides a set of capabilities through procedures and functions—with supporting constants, data types, and variables—but it hides how those capabilities are actually implemented by separating the unit into two sections: the *interface* and the *implementation*. When a program uses a unit, all the unit's declarations become available, as if they had been defined within the program itself.

A unit's structure is not unlike that of a program, but with some significant differences. Here's a unit, for example:

```
unit <identifier>;
interface
uses <list of units>;   { Optional }
  { public declarations }
implementation
uses <list of units>;   { Optional }
  { private declarations }
  { implementation of procedures and functions }
begin
  { initialization code }
end.
```

The unit header starts with the reserved word **unit**, followed by the unit's name (an identifier), much the way a program begins. The next item in a unit is the keyword **interface**. This signals the start of the interface section of the unit—the section visible to any other units or programs that use this unit.

A unit can use other units by specifying them in a **uses** clause. The **uses** clause can appear in two places. First, it can appear immediately after the keyword **interface**. In this case, any constants or data types declared in the interfaces of those units can be used in any of the declarations in this unit's interface section.

Second, it can appear immediately after the keyword **implementation**. In this case, any declarations from those units can be used only within the implementation section.

Interface section

The interface portion—the "public" part—of a unit starts at the reserved word **interface**, which appears after the unit header and ends when the reserved word **implementation** is encountered. The interface determines what is "visible" to any program (or other unit) using that unit; any program using the unit has access to these "visible" items.

In the unit interface, you can declare constants, data types, variables, procedures, and functions. As with a program, these can be arranged in any order, and sections can repeat themselves (for example, **type** ... **var** ... *<procs>* ... **const** ... **type** ... **const** ... **var**).

The procedures and functions visible to any program using the unit are declared here, but their actual bodies—their implementations—are found in the implementation section. **forward** declarations are neither necessary nor allowed. The bodies of all the regular procedures and functions are held in the implementation section after all the procedure and function headers have been listed in the interface section.

A **uses** clause may appear in the interface section. If present, **uses** must immediately follow the keyword **interface**.

Implementation section

The implementation section—the "private" part—starts at the reserved word **implementation**. Everything declared in the interface portion is visible in the implementation: constants, types, variables, procedures, and functions. Furthermore, the implementation can have additional declarations of its own, although these are not visible to any programs using the unit. The program doesn't know they exist and can't reference or call them. However, these hidden items can be (and usually are) used by the "visible" procedures and functions—those routines whose headers appear in the interface section.

A **uses** clause may appear in the implementation. If present, **uses** must immediately follow the keyword implementation.

If any procedures have been declared external, one or more {**$L** *filename*} directive(s) should appear anywhere in the source file before the final **end** of the unit so that the object file containing the procedures can be linked in.

The normal procedures and functions declared in the interface—those that are not inline—must reappear in the implementation. The **procedure/function** header that appears in the implementation should either be identical to that which appears in the interface or should be in the short form. For the short form, type in the keyword (**procedure** or **function**), followed by the routine's name (identifier). The routine will then contain all its local declarations (labels, constants, types, variables, and nested procedures and functions), followed by the main body of the routine itself. Say the following declarations appear in the interface of your unit:

```
procedure ISwap(var V1,V2: Integer);
function IMax(V1,V2: Integer): Integer;
```

The implementation could look like this:

```
procedure ISwap;
var
  Temp: Integer;
begin
   Temp := V1; V1 := V2; V2 := Temp;
end;  { of proc ISwap }
function IMax(V1, V2: Integer): Integer;
begin
  if V1 > V2 then IMax := V1
  else IMax := V2;
end;  { of func IMax }
```

Routines local to the implementation (that is, not declared in the interface section) must have their complete **procedure/function** header intact.

Initialization section

The entire implementation portion of the unit is normally bracketed within the reserved words **implementation** and **end**. However, if you put the reserved word **begin** before **end**, with statements between the two, the resulting compound statement—looking very much like the main body of a program—becomes the **initialization** section of the unit.

The initialization section is where you initialize any data structures (variables) that the unit uses or makes available (through the interface) to the program using it. You can use it to open files for the program to use later.

When a program using that unit is executed, the unit's initialization section is called before the program's main body is run. If the program uses more than one unit, each unit's initialization section is called (in the order specified in the program's **uses** statement) before the program's main body is executed.

How are units used?

The units your program uses have already been compiled and stored as machine code, not Pascal source code; they are not Include files. Even the interface section is stored in the special binary symbol table format that Turbo Pascal uses. Furthermore, certain standard units are stored in a special file (TPW.TPL) and are automatically loaded into memory along with Turbo Pascal itself.

As a result, using a unit or several units adds very little time (typically less than a second) to the length of your program's compilation. If the units are being loaded in from a separate disk file, a few additional seconds may be required because of the time it takes to read from the disk.

As stated earlier, to use a specific unit or collection of units, you must place a **uses** clause at the start of your program, followed by a list of the unit names you want to use, separated by commas:

```
program MyProg;
uses thisUnit, thatUnit, theOtherUnit;
```

When the compiler sees this **uses** clause, it adds the interface information in each unit to the symbol table and links the machine code that is the implementation to the program itself.

The ordering of units in the **uses** clause is not important. If *thisUnit* uses *thatUnit* or vice versa, you can declare them in either order, and the compiler will determine which unit must be linked into MyProg first. In fact, if *thisUnit* uses *thatUnit* but MyProg doesn't need to directly call any of the routines in *thatUnit*, you

can "hide" the routines in *thatUnit* by omitting it from the **uses** clause:

```
unit thisUnit;
uses thatUnit;
...
program MyProg;
uses thisUnit, theOtherUnit;
...
```

In this example, *thisUnit* can call any of the routines in *thatUnit*, and MyProg can call any of the routines in *thisUnit* or *theOtherUnit*. MyProg cannot, however, call any of the routines in *thatUnit* because *thatUnit* does not appear in MyProg's **uses** clause.

If you don't put a **uses** clause in your program, Turbo Pascal links in the *System* standard unit anyway. This unit provides some of the standard Pascal routines as well as a number of Turbo Pascal-specific routines.

Referencing unit declarations

Once you include a unit in your program, all the constants, data types, variables, procedures, and functions declared in that unit's interface become available to you. For example, suppose the following unit existed:

```
unit MyStuff;
interface
const
  MyValue  = 915;
type
  MyStars  = (Deneb, Antares, Betelgeuse);
var
  MyWord: string[20];
procedure SetMyWord(Star: MyStars);
function  TheAnswer: Integer;
implementation
...
end.
```

What you see here is the unit's interface, the portion that is visible to (and used by) your program. Given this, you might write the following program:

```
program TestStuff;
uses WinCrt, MyStuff;
```

```
var
  I: Integer;
  AStar: MyStars;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.
```

Now that you have included the statement **uses** *MyStuff* in your program, you can refer to all the identifiers declared in the interface section in the interface of *MyStuff* (*MyWord, MyValue,* and so on). However, consider the following situation:

```
program TestStuff;
uses WinCrt, MyStuff;
const
  MyValue = 22;
var
  I: Integer;
  AStar: MyStars;

function TheAnswer: Integer;
begin
  TheAnswer := -1;
end;

begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.
```

This program redefines some of the identifiers declared in *MyStuff*. It will compile and run, but will use its own definitions for *MyValue* and *TheAnswer*, since those were declared more recently than the ones in *MyStuff*.

You're probably wondering whether there's some way in this situation to still refer to the identifiers in *MyStuff*? Yes, preface each one with the identifier *MyStuff* and a period (.). For example, here's yet another version of the earlier program:

```
program TestStuff;
uses WinCrt, MyStuff;
const
  MyValue = 22;
var
  I: Integer;
  AStar: MyStars;

function TheAnswer: Integer;
begin
  TheAnswer := -1;
end;

begin
  Writeln(MyStuff.MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := MyStuff.TheAnswer;
  Writeln(I);
end.
```

This program will give you the same answers as the first one, even though you've redefined *MyValue* and *TheAnswer*. Indeed, it would have been perfectly legal (although rather wordy) to write the first program as follows:

```
program TestStuff;
uses WinCrt, MyStuff;
var
  I: Integer;
  AStar: MyStuff.MyStars;
begin
  Writeln(MyStuff.MyValue);
  AStar := MyStuff.Deneb;
  MyStuff.SetMyWord(AStar);
  Writeln(MyStuff.MyWord);
  I := MyStuff.TheAnswer;
  Writeln(I);
end.
```

Note that you can preface any identifier—constant, data type, variable, or subprogram—with the unit name.

Implementation section uses clause

Turbo Pascal for Windows allows you to place a **uses** clause in a unit's implementation section. If present, the **uses** clause must immediately follow the **implementation** keyword, just like a **uses**

clause in the interface section must immediately follow the **interface** keyword.

A **uses** clause in the implementation section allows you to further hide the inner details of a unit, since units used in the implementation section are not visible to users of the unit. More importantly, however, it also enables you to construct mutually dependent units.

Since units in Turbo Pascal need not be strictly hierarchical, you can make circular unit references (for more about this, read Chapter 9 in the *Programmer's Guide*).

The standard units

The file TPW.TPL contains all the standard units : *System*, *WinDos*, *WinProcs*, *WinTypes*, *Strings*, and *WinCrt*. These are units loaded into memory with Turbo Pascal; they're always readily available to you. You will normally keep the file TPW.TPL in the same directory as TPW.EXE (or TPCW.EXE).

System

The details of the System unit are described in Chapter 11 of the Programmer's Guide, "The System unit."

System contains all the standard and built-in procedures and functions of Turbo Pascal. Every Turbo Pascal routine that is *not* part of standard Pascal and that is *not* in one of the other units is in *System*. This unit is always linked into every program.

WinDos

The WinDos unit is discussed in detail in Chapter 12 of the Programmer's Guide.

WinDos defines numerous Pascal procedures and functions that are equivalent to the most commonly used DOS calls, such as *GetTime*, *SetTime*, *DiskSize*, and so on. It also defines two low-level routines, *MsDos* and *Intr*, which allow you to directly invoke any MS-DOS call or system interrupt. *TRegisters* is the data type for the parameter to *MsDos* and *Intr*. Some other constants and data types are also defined.

Strings

The Strings unit is discussed in detail in Chapter 13 of the Programmer's Guide.

The *Strings* unit handles the new null-terminated strings. The *System* unit still processes Pascal-style strings.

WinCrt

The WinCrt unit is discussed in detail in Chapter 14 in the Programmer's Guide.

The *WinCrt* unit is a text file device driver that redirects your program's output to a scrollable window. Although most of your Turbo Pascal for Windows programs will create their own windows, you can use the *WinCrt* unit for quick and simple text-based programs.

WinTypes

See the Windows Reference Guide for detailed information on WinTypes.

The *WinTypes* unit contains all the constants, data structures, and styles in the Windows API.

WinProcs

See the Windows Reference Guide for detailed information on WinProcs.

The *WinProcs* unit contains all the functions and procedures that make up the Windows API.

Now that you've been introduced to units, let's see about writing your own.

Writing your own units

Say you've written a unit called *IntLib*, stored it in a file called INTLIB.PAS, and compiled it to disk; the resulting code file will be called INTLIB.TPU. To use it in your program, you must include a **uses** statement to tell the compiler you're using that unit. Your program might look like this:

```
program MyProg;
uses IntLib;
```

Note that Turbo Pascal expects the unit code file to have the same name (up to eight characters) of the unit itself. If your unit name is *MyUtilities*, then Turbo is going to look for a file called MYUTILIT.TPU.

Compiling units

You compile a unit exactly the way you'd compile a program: Write it using the editor and select the Compile | Compile command (or press *Alt+F9*). Instead of creating an .EXE file, however, Turbo Pascal will create a .TPU (**T**urbo **P**ascal **U**nit) file.

You can then leave this file as is or merge it into TPW.TPL using TPUMOVER.EXE.

The compiler searches for units in all the paths specified in the Unit Directories input box.

In any case, you probably want to copy your .TPU files (along with their source) to the unit directory you specified in the Unit Directories input box (Options | Directories). That way, you can reference those files without having to have them in the current work directory or in TPW.TPL. (See page 114 in Chapter 5, "Project management" for more information about how Turbo Pascal determines the current work directory.)

You can only have one unit in a given source file; compilation stops when the final **end** statement is encountered.

To locate a unit specified in a **uses** clause, the compiler first checks the resident units—those units loaded into memory at startup from the TPW.TPL file. If the unit is not among the resident units, the compiler assumes it must be on disk. The name of the file is assumed to be the unit name with extension .TPU. It is first searched for in the current work directory, and then in the directories specified with the Options | Directories | Unit command or in a **/U** directive on the TPCW command line. For instance, the construct

```
uses Memory;
```

where *Memory* is not a resident unit, causes the compiler to look for the file MEMORY.TPU in the current work directory, and then in each of the unit directories.

When the Compile | Make and Compile | Build commands compile the units specified in a **uses** clause, the source files are searched for in the same way as the .TPU files, and the name of a given unit's source file is assumed to be the unit name with extension .PAS.

An example

Okay, now let's write a small unit. We'll call it *IntLib* and put in two simple integer routines—a procedure and a function:

```
unit IntLib;
interface
procedure ISwap(var I,J: Integer);
function  IMax(I,J: Integer): Integer;
implementation
procedure ISwap;
var
```

```
  Temp: Integer;
begin
  Temp := I; I := J; J := Temp;
end;  { of proc ISwap }
function IMax;
begin
  if I > J then
    IMax := I
  else IMax := J;
end;  { of func IMax }
end.  { of unit IntLib }
```

Type this in, save it as the file INTLIB.PAS, then compile it. The resulting unit code file is INTLIB.TPU. Move it to your unit directory (whevever that might happen to be) or leave it in the same directory as the program that follows. This next program uses the unit *IntLib*:

```
program IntTest;
uses WinCrt, IntLib;
var
  A, B: Integer;
begin
  Write('Enter two integer values:  ');
  Readln(A, B);
  ISwap(A, B);
  Writeln('A = ', A, ' B = ', B);
  Writeln('The max is ', IMax(A, B));
end.  { of program IntTest }
```

Congratulations! You've just created your first unit and written a program that uses it!

Units and large programs

Up until now, you've probably thought of units only as libraries—collections of useful routines to be shared by several programs. Another function of a unit, however, is to break up a large program into modules.

Two aspects of Turbo Pascal make this modular functionality of units work: (1) its tremendous speed in compiling and linking and (2) its ability to manage several code files simultaneously, such as a program and several units.

Typically, a large program is divided into units that group procedures by their function. For instance, an editor application could be divided into initialization, printing, reading and writing

files, formatting, and so on. Also, there could be a "global" unit—one used by all other units, as well as the main program—that defines global constants, data types, variables, procedures, and functions.

The skeleton of a large program might look like this:

```
program Editor;
uses
  WinCrt, Strings,                    { Standard units from TPW.TPL }
  EditGlobals,                                { User-written units }
  EditInit,
  EditPrint,
  EditRead, EditWrite,
  EditFormat;

{ Program's declarations, procedures, and functions }
begin  { main program }
end.  { of program Editor }
```

Note that the units in this program could either be in TPW.TPL or in their own individual .TPU files. If the latter is true, then Turbo Pascal will manage your project for you. This means when you recompile program *Editor* using the compiler's built-in make facility, Turbo Pascal will compare the dates of each .PAS and .TPU file and recompile modules whose source has been modified.

See Chapter 5, "Project management," for more information about how to deal with large programs.

Another reason to use units in large programs has to do with code segment limitations. The 8086 (and related) processors limit the size of a given chunk, or segment, of code to 64K. This means that the main program and any given segment cannot exceed a 64K size. Turbo Pascal handles this by making each unit a separate code segment. Without units, you're limited to 64K of code for your program.

The TPUMOVER utility

Suppose you want to add a well-designed and thoroughly debugged unit to the library of standard units (TPW.TPL) so that it's automatically loaded into memory when you run the compiler. Is there any way to add to TPW.TPL? Yes, by using the TPUMOVER.EXE utility.

You can also use TPUMOVER to remove units from the Turbo Pascal standard unit library file, reducing its size and the amount of memory it takes up when loaded.

As you've seen, it's really quite simple to write your own units. A well-designed, well-implemented unit simplifies program development; you solve the problems only once, not for each new program. Best of all, a unit provides a clean, simple mechanism for writing very large programs.

C H A P T E R

4

Object-oriented programming

Object-oriented programming (OOP) is a method of programming that closely mimics the way all of us get things done. It is a natural evolution from earlier innovations to programming language design: It is more structured than previous attempts at structured programming; and it is more modular and abstract than previous attempts at data abstraction and detail hiding. Three main properties characterize an object-oriented programming language:

- *Encapsulation*: Combining a record with the procedures and functions that manipulate it to form a new data type—an object.
- *Inheritance*: Defining an object and then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors' code and data.
- *Polymorphism*: Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

Turbo Pascal's language extensions give you the full power of object-oriented programming: more structure and modularity, more abstraction, and reusability built right into the language. All these features add up to code that is more structured, extensible, and easy to maintain.

The challenge of object-oriented programming is that it requires you to set aside habits and ways of thinking about programming that have been standard for many years. Once you do that,

however, OOP is a simple, straightforward, superior tool for solving many of the problems that plague traditional programs.

A note to you who have done object-oriented programming in other languages: Put aside your previous impressions of OOP and learn Turbo Pascal's object-oriented features on their own terms. OOP is not one single way of programming; it is a continuum of ideas. In its object philosophy, Turbo Pascal is more like C++ than Smalltalk. Smalltalk is an interpreter, while from the beginning, Turbo Pascal has been a pure native code compiler. Native code compilers do things differently (and far more quickly) than interpreters.

And a note to you who haven't any notion at all what OOP is about: That's just as well. Too much hype, too much confusion, and too many people talking about something they don't understand have greatly muddied the waters in recent years. Strive to forget what people have told you about OOP. The best way (in fact, the *only* way) to learn anything useful about OOP is to do what you're about to do: Sit down and try it yourself.

Objects?

Yes, objects. Look around you...there's one: the apple you brought in for lunch. Suppose you were going to describe an apple in software terms. The first thing you might be tempted to do is pull it apart: Let *S* represent the area of the skin; let *J* represent the fluid volume of juice it contains; let *F* represent the weight of fruit inside; let *D* represent the number of seeds....

Don't think that way. Think like a painter. You see an apple, and you paint an apple. The picture of an apple is not an apple; it's just a symbol on a flat surface. But it hasn't been abstracted into seven numbers, all standing alone and independent in a data segment somewhere. Its components remain together, in their essential relationships to one another.

Objects model the characteristics and behavior of the elements of the world we live in. They are the ultimate data abstraction so far.

Objects keep all their characteristics and behavior together.

An apple can be pulled apart, but once it's been pulled apart it's not an apple anymore. The relationships of the parts to the whole and to one another are plainer when everything is kept together in one wrapper. This is called *encapsulation*, and it's very important. We'll return to encapsulation in a little while.

Equally important, objects can *inherit* characteristics and behavior from what are called *ancestor objects*. This is an intuitive leap; inheritance is perhaps the single biggest difference between object-oriented Turbo Pascal and Standard Pascal programming today.

Inheritance

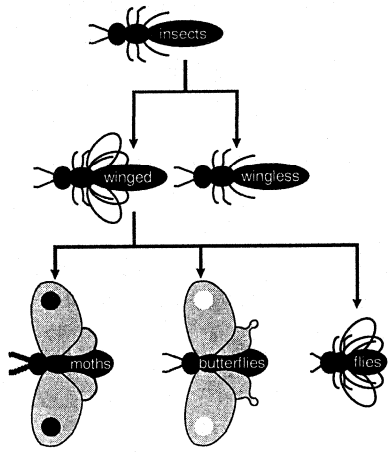
The goal of science is to describe the workings of the universe. Much of the work of science, in furthering that goal, is simply the creation of family trees. When entomologists return from the Amazon with a previously unknown insect in a jar, their fundamental concern is working out where that insect fits into the giant chart upon which the scientific names of all other insects are gathered. There are similar charts of plants, fish, mammals, reptiles, chemical elements, subatomic particles, and external galaxies. They all look like family trees: a single overall category at the top, with an increasing number of categories beneath that single category, fanning out to the limits of diversity.

Within the category *insect*, for example, there are two divisions: insects with visible wings, and insects with hidden wings or no wings at all. Under winged insects is a larger number of categories: moths, butterflies, flies, and so on. Each category has numerous subcategories, and beneath those subcategories are even more subcategories (see Figure 4.1).

This classification process is called *taxonomy*. It's a good starting metaphor for the inheritance mechanism of object-oriented programming.

The questions a scientist asks in trying to classify a new animal or object are these: *How is it similar to the others of its general class? How is it different?* Each different class has a set of behaviors and characteristics that define it. A scientist begins at the top of a specimen's family tree and starts descending the branches, asking those questions along the way. The highest levels are the most general, and the questions the simplest: Wings or no wings? Each level is more specific than the one before it, and less general. Eventually the scientist gets to the point of counting hairs on the third segment of the insect's hind legs—specific indeed (and a good reason, perhaps, not to be an entomologist).

Figure 4.1
A partial taxonomy chart of insects



The important point to remember is that once a characteristic is defined, all the categories *beneath* that definition *include* that characteristic. So once you identify an insect as a member of the order *diptera* (flies), you needn't make the point that a fly has one pair of wings. The species of insect called *flies* inherits that characteristic.

As you'll learn shortly, object-oriented programming is the process of building family trees for data structures. One of the important things object-oriented programming adds to traditional languages like Pascal is a mechanism by which data types inherit characteristics from simpler, more general types. This mechanism is inheritance.

Objects: records that inherit

In Pascal terms, an object is very much like a record, which is a wrapper for joining several related elements of data together under one name. Suppose you want to develop a payroll program that produces a report showing how much each employee gets paid each payday. You might lay out a record like this:

We start all our type names with the letter T. This is a convention you may want to follow.

```
TEmployee = record
  Name: string[25];
  Title: string[25];
  Rate: Real;
end;
```

TEmployee here is a *record type*; that is, it's a template that the compiler uses to create record variables. A variable of type *TEmployee* is an instance of type *TEmployee*. The term *instance* is used now and then in Pascal circles, but it is used all the time by OOP people, and you'll do well to start thinking in terms of types and instances of those types.

With type *TEmployee* you have it both ways: You can think of the *Name*, *Title*, and *Rate* fields separately, or when you need to think of the fields working together to describe a particular worker, you can think of them collectively as *TEmployee*.

Suppose you have several types of employees working in your company. Some are paid hourly, some are salaried, some are commissioned, and so on. Your payroll program needs to accommodate all these types. You might develop a different record type for each type of employee. For example, to figure out how much an hourly employee gets paid, you need to know how many hours the employee worked. You could design a *THourly* record like this:

```
THourly = record
  Name: string[25];
  Title: string[25];
  Rate: Real;
  Time: Integer;
end;
```

You might also be a little more clever and retain record type *TEmployee* by creating a field of type *TEmployee* within type *THourly*:

```
THourly = record
  Worker: TEmployee;
  Time: Integer;
end;
```

This works, and Pascal programmers do it all the time. One thing this method doesn't do is force you to think about the nature of what you're manipulating in your software. You need to ask questions like, "How does an hourly employee differ from other employees?" The answer is this: An hourly employee is a employee who is paid for the number of hours the employee works. Think back on the first part of that statement: An *hourly* employee is an *employee*....

There you have it!

An *hourly* employee record must have all the fields that exist in the *employee* record. Type *THourly* is a descendant type of type *TEmployee*. *THourly* inherits everything *TEmployee* has, and adds whatever is new about *THourly* to make *THourly* unique.

This process by which one type inherits the characteristics of another type is called *inheritance*. The inheritor is called a *descendant type*; the type that the descendant type inherits from is an *ancestor type*.

The familiar Pascal record types cannot inherit. Turbo Pascal, however, extends the Pascal language to support inheritance. One of these extensions is a new category of data structure, related to records but far more powerful. Data types in this new category are defined with a new reserved word: **object**. An object type can be defined as a complete, stand-alone type in the fashion of Pascal records, or it can be defined as a descendant of an existing object type by placing the name of the ancestor type in parentheses after the reserved word **object**.

In the payroll example you just looked at, the two related object types would be defined this way:

```
type
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate: Real;
  end;

  THourly = object(TEmployee)
    Time: Integer;
  end;
```

Note the use of parentheses here to denote inheritance.

Here, *TEmployee* is the ancestor type, and *THourly* is the descendant type. As you'll see a little later, the process can continue indefinitely: You can define descendants of type *THourly*, and descendants of *THourly*'s descendant type, and so on. A large part of designing an object-oriented application lies in building this *object hierarchy* expressing the family tree of the objects in the application.

All the types eventually inheriting from *TEmployee* are called *TEmployee*'s descendant types, but *THourly* is one of *TEmployee*'s *immediate descendants*. Conversely, *TEmployee* is *THourly*'s *immediate ancestor*. An object type (just like a DOS subdirectory) can have any number of immediate descendants, but only one immediate ancestor.

Objects are closely related to records, as these definitions show. The new reserved word **object** is the most obvious difference, but there are numerous other differences, some of them quite subtle, as you'll see later.

For example, the *Name*, *Title*, and *Rate* fields of *TEmployee* are not explicitly written into type *THourly*, but *THourly* has them anyway, by virtue of inheritance. You can speak about *THourly*'s *Name* value, just as you can speak about *TEmployee*'s *Name* value.

Instances of object types

Instances of object types are declared just as any variables are declared in Pascal, either as static variables or as pointer referents allocated on the heap:

```
type
  PHourly = ^THourly;

var
  StatHourly: THourly;  { Ready to go! }
  DynaHourly: PHourly;  { Must allocate with New before use }
```

An object's fields

You access an object's data fields just as you access the fields of an ordinary record, either through the **with** statement or by *dotting*; for example,

```
AnHourly.Rate := 9.45;

with AnHourly do
begin
  Name  := 'Sanderson, Arthur';
  Title := 'Word processor';
end;
```

Don't forget: An object's inherited fields are not treated specially simply because they are inherited.

You just have to remember at first (eventually it comes naturally) that inherited fields are just as accessible as fields declared within a given object type. For example, even though *Name*, *Title*, and *Rate* are not part of *THourly*'s declaration (they are inherited from type *TEmployee*), you can specify them just as though they were declared within *THourly*:

```
AnHourly.Name := 'Arthur Sanderson';
```

Good practice and bad practice

Turbo Pascal actually lets you make an object's fields and methods private; for more on this, refer to page 80.

Even though you can access an object's fields directly, it's not an especially good idea to do so. Object-oriented programming principles require that an object's fields be left alone as much as possible. This restriction might seem arbitrary and rigid at first, but it's part of the big picture of OOP that is being built in this chapter. In time you'll see the sense behind this new definition of good programming practice, though there's some ground to cover before it all comes together. For now, take it on faith: Avoid accessing object data fields directly.

So—how are object fields accessed? What sets them and reads them?

An object's data fields are what an object knows; its methods are what an object does.

The answer is that an object's *methods* are used to access an object's data fields whenever possible. A *method* is a procedure or function declared *within* an object and tightly bonded to that object.

Methods

Methods are one of object-oriented programming's most striking attributes, and they take some getting used to. Start by harkening back to that fond old necessity of structured programming, initializing data structures. Consider the task of initializing a record with this definition:

```
Employee = record
  Name: string[25];
  Title: string[25];
  Rate: Real;
end;
```

Most programmers would use a **with** statement to assign initial values to the *Name*, *Title*, and *Rate* fields:

```
var
  MyEmployee: Employee;

with MyEmployee do
begin
  Name := 'Arthur Sanderson';
  Title := 'Word processor';
  Rate := 9.45;
end;
```

This works well, but it's tightly bound to one specific record instance, *MyEmployee*. If more than one *Employee* record needs to be initialized, you'll need more **with** statements that do essentially the same thing. The natural next step is to build an initialization procedure that generalizes the **with** statement to encompass any instance of a *TEmployee* type passed as a parameter:

```
procedure InitTEmployee(var Worker: TEmployee; AName,
  ATitle: String; ARate: Real);
begin
  with Worker do
  begin
    Name  := NewName;
    Title := NewTitle;
    Rate  := NewRate;
  end;
end;
```

This does the job, all right—but if you're getting the feeling that it's a little more fooling around than it ought to be, you're feeling the same thing that object-oriented programming's early proponents felt.

It's a feeling that implies that, well, you've designed procedure *InitTEmployee* specifically to serve type *TEmployee*. Why, then, must you keep specifying what record type and instance *InitTEmployee* acts upon? There should be some way of welding together the record type and the code that serves it into one seamless whole.

Now there is. It's called a *method*. A method is a procedure or function welded so tightly to a given type that the method is surrounded by an invisible **with** statement, making instances of that type accessible from within the method. The type definition includes the header of the method. The full definition of the method is qualified with the name of the type. Object type and object method are the two faces of this new species of structure called an object:

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
    procedure Init(NewName, NewTitle: string[25];
      NewRate: Real);
  end;
```

```
procedure TEmployee.Init(NewName, NewTitle: string[25]; NewRate:
  Real);
begin
  Name  := NewName;  { The Name field of an TEmployee object }
  Title := NewTitle; { The Title field of an TEmployee object }
  Rate  := NewRate;  { The Rate field of an TEmployee object }
end;
```

Now, to initialize an instance of type *TEmployee*, you simply call its method as though the method were a field of a record, which in one very real sense it is:

```
var
  AnEmployee: TEmployee;

AnEmployee.Init('Sara Adams, Account manager, 15000');  { Easy, no? }
```

Code and data together

One of the most important tenets of object-oriented programming is that the programmer should think of code and data *together* during program design. Neither code nor data exists in a vacuum. Data directs the flow of code, and code manipulates the shape and values of data.

When your data and code are separate entities, there's always the danger of calling the right procedure with the wrong data or the wrong procedure with the right data. Matching the two is the programmer's job, and while Pascal's strong typing does help, at best it can only say what *doesn't* go together.

Pascal says nothing, anywhere, about what *does* go together. If it's not in a comment or in your head, you take your chances.

By bundling code and data declarations together, an object helps keep them in sync. Typically, to get the value of one of an object's fields, you call a method belonging to that object that returns the value of the desired field. To set the value of a field, you call a method that assigns a new value to that field.

See "Private section" on page 80 for details on how to do this.

Like many aspects of object-oriented programming, respect for encapsulated data is a discipline you should always observe. It's better to access an object's data by using the methods it provides, instead of reading the data directly. Turbo Pascal lets you enforce encapsulation through the use of a **private** declaration in an object's declaration.

Defining methods

The process of defining an object's methods is reminiscent of Turbo Pascal units. Inside an object, a method is defined by the header of the function or procedure acting as a method:

All data fields must be declared before the first method declaration.

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
    procedure Init(AName, ATitle: String; ARate: Real);
    function GetName : String
    function GetTitle: String;
    function GetRate : Real;
  end;
```

As with procedure and function declarations in a unit's **interface** section, method declarations within an object tell *what* a method does, but not *how*.

The *how* is defined *outside* the object definition, in a separate procedure or function declaration. When methods are fully defined outside the object, the name of the object type that owns the method, followed by a period, must precede the method name:

```
procedure TEmployee.Init(AName, ATitle: String;
  ARate: Real);
begin
  Name  := AName;
  Title := ATitle;
  Rate  := ARate;
end;

function TEmployee.GetName: String;
  GetName := Name;
end;

function TEmployee.GetTitle: String;
begin
  GetTitle := Title;
end;

function TEmployee.GetRate: Real;
begin
  GetRate := Rate;
end;
```

Method definition follows the intuitive dotting method of specifying a record field. In addition to having a definition of *TEmployee.GetName*, it would be completely legal to define a procedure named *GetName* without the identifier *TEmployee* preceding it. However, the "outside" *GetName* would have no connection to the object type *TEmployee* and would probably confuse the sense of the program as well.

Method scope and the Self parameter

Notice that nowhere inside a method is there an explicit **with** `object` **do**... construct. The data fields of an object are freely available to that object's methods. Although they are separated in the source code, the method bodies and the object's data fields really share the same scope.

This is why one of *TEmployee*'s methods can contain the statement `GetTitle := Title` without any qualifier to *Title*. It's because *Title belongs to the object that called the method.* When an object calls a method, there is an implicit statement to the effect **with** `myself` **do** `method` linking the object and its method in scope.

This implicit **with** statement is accomplished by the passing of an invisible parameter to the method each time any method is called. This parameter is called *Self*, and is actually a full 32-bit pointer to the object instance making the method call. The *GetRate* method belonging to *TEmployee* is roughly equivalent to the following:

This example is not fully correct syntactically; it's here simply to give you a fuller appreciation for the special link between an object and its methods.

```
function TEmployee.GetRate(var Self: TEmployee): Integer;
begin
  GetRate := Self.Rate;
end;
```

Is it important for you to be aware of *Self*? Ordinarily, no: Turbo Pascal's generated code handles it automatically in virtually all cases. There are a few circumstances, however, when you might have to intervene inside a method and make explicit use of the *Self* parameter.

For more details on method call stack frames, see Chapter 18 in the Programmer's Guide.

The *Self* parameter is part of the physical stack frame for all method calls. Methods implemented as externals in assembly language must take *Self* into account when they access method parameters on the stack.

Object data fields and method formal parameters

One consequence of the fact that methods and their objects share the same scope is that a method's formal parameters cannot be identical to any of the object's data fields. This is not some new restriction imposed by object-oriented programming, but rather the same old scoping rule that Pascal has always had. It's the same as not allowing the formal parameters of a procedure to be identical to the procedure's local variables:

```
procedure CrunchIt(Crunchee: MyDataRec, Crunchby, ErrorCode:
  Integer);
var
  A, B: Char;
  ErrorCode: Integer; { This declaration causes an error! }
begin
  ...
```

A procedure's local variables and its formal parameters share the same scope and thus cannot be identical. You'll get "Error 4: Duplicate identifier" if you try to compile something like this; the same error occurs if you attempt to give a method a formal parameter identical to any field in the object that owns the method.

The circumstances are a little different, since having procedure headers inside a data structure is a wrinkle new to Turbo Pascal, but the guiding principles of Pascal scoping have not changed at all.

Objects exported by units

It makes good sense to define objects in units, with the object type declaration in the interface section of the unit and the procedure bodies of the object type's methods defined in the implementation section.

By "exported" we mean "defined within the interface section of a unit."

Units can have their own private object type definitions in the implementation section, and such types are subject to the same restrictions as any types defined in a unit implementation section. An object type defined in the interface section of a unit can have descendant object types defined in the implementation section of the unit. In a case where unit *B* uses unit *A*, unit *B* can also define descendant types of any object type exported by unit *A*.

The object types and methods described earlier can be defined within a unit as shown in WORKERS.PAS on your disk. To make

use of the object types and methods defined in unit *Workers*, you simply use the unit in your own program, and declare an instance of type *THourly* in the **var** section of your program:

```
program HourRpt;

uses WinCrt, Workers;

var
  AnHourly: THourly;
  ...
```

To create and print the hourly employee's name, title, and amount of pay represented by *AnHourly*, you simply call *AnHourly*'s methods, using the dot syntax:

```
AnHourly.Init('Sara Adams', 'Account manager', 1400);
                      { Initializes an instance of THourly with }
                      { employee data for Sara Adams }
Show;                 { Writes name, title, and pay amount }
```

Objects can also be typed constants.

Objects, being very similar to records, can also be used inside **with** statements. In that case, naming the object that owns the method isn't necessary:

```
with AnHourly do
begin
  Init('Sara Adams', 'Account manager', 1400);
  Show;
end;
```

Just as with records, objects can be passed to procedures as parameters and (as you'll see later on) can also be allocated on the heap.

Private section

In some circumstances you may have parts of an object declaration that you don't want to export. For example, you may want to provide objects for other programmers to use without letting them manipulate the object's data directly. To make it easy for you, Turbo Pascal allows you to specify private fields and methods within objects.

Private fields and methods are accessible only within the unit in which the object is declared. In the previous example, if the type *THourly* had private fields, they could only be accessed by code within the *THourly* unit. Even though other parts of *THourly* would be exported, the parts declared as private would be inaccessible.

Private fields and methods are declared just after regular fields and methods, following the optional **private** reserved word. Thus, the full syntax for an object declaration is

```
type
  NewObject = object(ancestor)
    fields; { these are public }
    methods; { these are public }
  private
    fields; { these are private }
    methods; { these are private }
  end;
```

Programming in the active voice

Most of what's been said about objects so far has been from a comfortable, Turbo Pascal-ish perspective, since that's most likely where you are coming from. This is about to change, as you move on to OOP concepts with fewer precedents in standard Pascal programming. Object-oriented programming has its own particular mindset, due in part to OOP's origins in the (somewhat insular) research community, but also because the concept is truly and radically different.

Object-oriented languages were once called "actor languages" with this metaphor in mind.

One often amusing outgrowth of this is that OOP fanatics anthropomorphize their objects. Data structures are no longer passive buckets that you toss values into. In the new view of things, an object is looked upon as an actor on a stage, with a set of lines (methods) memorized. When you (the director) give the word, the actor recites from the script.

It can be helpful to think of the function *AnHourly.GetPayAmount* as giving an order to object *AnHourly*, saying "Calculate the amount of your pay check." The object is the central concept here. Both the list of methods and the list of data fields contained by the object serve the object. Neither code nor data is boss.

Objects aren't being described as actors on a stage just to be cute. The object-oriented programming paradigm tries very hard to model the components of a problem as components, and not as logical abstractions. The odds and ends that fill our lives, from toasters to telephones to terry towels, all have characteristics (data) and behaviors (methods). A toaster's characteristics might include the voltage it requires, the number of slices it can toast at once, the setting of the light/dark lever, its color, its brand, and so

on. Its behaviors include accepting slices of bread, toasting slices of bread, and popping toasted slices back up again.

If you wanted to write a kitchen simulation program, what better way to do it than to model the various appliances as objects, with their characteristics and behaviors encoded into data fields and methods? It's been done, in fact; the very first object-oriented language (Simula-67) was created as a language for writing such simulations.

This is the reason that object-oriented programming is so firmly linked in conventional wisdom to graphics-oriented environments. Objects in Turbo Pascal should model components of the problem you're trying to solve. Keep that in mind as you further explore Turbo Pascal's object-oriented extensions.

Encapsulation

*Declaring fields as **private** allows you to enforce access to those fields only through methods.*

The welding of code and data together into objects is called *encapsulation*. If you're thorough, you can provide enough methods so that a user of the object never has to access its fields directly. Like Smalltalk and other programming languages, Turbo Pascal lets you enforce encapsulation through the use of a **private** directive. In this example, we won't specify a **private** section for fields and methods, but instead we will restrict ourselves to using methods in order to access the data we want.

TEmployee and *THourly* are written such that it is completely unnecessary to access any of their internal data fields directly:

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
    procedure Init(AName, ATitle: String; ARate: Real);
    function GetName: String;
    function GetTitle: String;
    function GetRate: Real;
    function GetPayAmount: Real;
  end;
```

```
THourly = object(TEmployee)
  Time: Integer;
  procedure Init(AName, ATitle: String; ARate: Real; ATime:
    Integer);
  function GetPayAmount: Real;
end;
```

There are only four data fields here: *Name*, *Title*, *Rate*, and *Time*. The *ShowName* and *ShowTitle* methods print an employee's name and title respectively. *GetPayAmount* uses *Rate*, and, in the case of an *THourly* employee, *Time*, to calculate the employee's pay check amount. There is no further need to access these data fields directly.

Assuming an instance of type *THourly* called *AnHourly*, you would use this suite of methods to manipulate *AnHourly*'s data fields indirectly, like this:

```
with AnHourly do
begin
  Init('Allison Karlon, Fork lift operator', 12.95, 62);
  Show;              { Writes name, title, and pay amount to screen }
end;
```

Note that the object's fields are not accessed at all except by the object's methods.

Methods: no downside

Adding these methods bulks up *THourly* a little in source form, but the Turbo Pascal smart linker strips out any method code that is never called in a program. You therefore shouldn't hang back from giving an object type a method that might or might not be used in every program that uses the object type. Unused methods cost you nothing in performance or .EXE file size—if they're not used, they're simply not there.

About data abstraction

There are powerful advantages to being able to completely decouple *THourly* from global references. If nothing outside the object "knows" the representation of its internal data, the programmer who controls the object can alter the details of the internal data representation—as long as the method headers remain the same.

Within some object, data might be represented as an array, but later on (perhaps as the scope of the application grows and its data volume expands), a binary tree might be recognized as a more efficient representation. If the object is completely encapsu-

lated, a change in data representation from an array to a binary tree *does not alter the object's use at all.* The interface to the object remains completely the same, allowing the programmer to fine-tune an object's performance without breaking any code that uses the object.

Extending objects

People who first encounter Pascal often take for granted the flexibility of the standard procedure *Writeln*, which allows a single procedure to handle parameters of many different types:

```
Writeln(CharVar);       { Outputs a character value }
Writeln(IntegerVar);    { Outputs an integer value }
Writeln(RealVar);       { Outputs a floating-point value }
```

Unfortunately, standard Pascal has no provision for letting you create equally flexible procedures of your own.

Object-oriented programming solves this problem through inheritance: When a descendant type is defined, the methods of the ancestor type are inherited, but they can also be overridden if desired. To override an inherited method, simply define a new method with the same name as the inherited method, but with a different body and (if necessary) a different set of parameters.

A simple example should make both the process and the implications clear. We have already defined a descendant type to *TEmployee* that represents an employee that is paid hourly wages:

```
const
  PayPeriods = 26;          { per annum }
  OvertimeThreshold = 80    { per pay period }
  OvertimeFactor = 1.5      { times normal hourly rate }

type
  THourly = object(TEmployee)
    Time: Integer;
    procedure Init(AName, ATitle: String; ARate: Real; ATime:
       Integer);
    function GetPayAmount: Real;
  end;

procedure THourly.Init(AName, ATitle: String; ARate: Real;
    ATime: Integer);
begin
  THourly.Init(AName, ATitle, ARate);
  Time := ATime;
end;
```

```
procedure THourly.GetPayAmount: Real;
var
  Overtime· Integer;
begin
  Overtime := Time - OvertimeThreshold;
  if Overtime > 0 then
     GetPayAmount := RoundPay(OvertimeThreshold * Rate
       + Rate OverTime * OvertimeFactor * Rate)
  else
     GetPayAmount := RoundPay(Time * Rate);
end;
```

A person who is paid hourly wages is still an employee: That person has everything we used to define the *TEmployee* object (name, title, rate of pay) except that the amount of money an hourly employee is paid depends on how many hours that employee has worked during a pay period. Therefore, *THourly* requires another field, *Time*.

Since *THourly* defines a new field, *Time*, initializing it requires a new *Init* method that initializes *Time* as well as the inherited fields. Rather than directly assigning values to inherited fields like *Name*, *Title* and *Rate*, why not reuse *TEmployee*'s initialization method (illustrated by *THourly.Init*'s first statement)? The syntax for calling an inherited method is *Ancestor.Method*, where *Ancestor* is the type identifier of an ancestral object type, and *Method* is a method identifier of that type.

Note that calling the method you override is not merely good style; it's entirely possible that *TEmployee.Init* performs some important, hidden initialization. By calling the overridden method, you ensure that the descendant object type includes its ancestor's functionality. In addition, any changes made to the ancestor's method automatically affects all its descendants.

After calling *TEmployee.Init*, *THourly.Init* can then perform its own initialization, which in this case consists only of assigning *Time* the value passed in *ATime*.

The *THourly.GetPayAmount* function, which calculates the amount an hourly employee is paid, is another example of an overriding method. In fact, each type of employee object has its own *GetPayAmount* method, because how the employee's pay amount is calculated differs depending on the employee type. The *THourly.GetPayAmount* method must consider how many hours the employee worked, if the employee worked overtime, what the overtime pay factor is, and so on. The *TSalaried.GetPayAmount*

method needs only to divide an employee's rate of pay by the number of pay periods per year (26 in our example).

```
unit Workers;

interface

const
  PayPeriods = 26;              { per annum }
  OvertimeThreshold = 80;       { per pay period }
  OvertimeFactor = 1.5;         { times normal hourly rate }

type
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate: Real;
    procedure Init(AName, ATitle: String; ARate: Real);
    function GetName: String;
    function GetTitle: String;
    function GetPayAmount: Real;
  end;

  THourly = object(TEmployee)
    Time: Integer;
    procedure Init(AName, ATitle: String; ARate: Real; ATime:
      Integer);
    function GetPayAmount: Real;
    function GetTime: Integer;
  end;

  TSalaried = object(TEmployee)
    function GetPayAmount: Real;
  end;

  TCommissioned = object(TSalaried)
    Commission: Real;
    SalesAmount: Real;
    constructor Init(AName, ATitle: String; ARate, ACommission,
      ASalesAmount: Real);
    function GetPayAmount: Real;
  end;

implementation

function RoundPay(Wages: Real): Real;
{ Round pay amount to ignore any pay less than 1 penny }
begin
  RoundPay := Trunc(Wages * 100) / 100;
end;
.
.
.
```

TEmployee is at the top of our object hierarchy and it contains the first *GetPayAmount* method.

```
function TEmployee.GetPayAmount: Real;
begin
  RunError(211);          { Give runtime error }
end;
```

You may wonder why all this method does is give you a run-time error. If *TEmployee.GetPayAmount* is called, an error exists in your program. Why? Because *TEmployee* is just the top of our object hierarchy and doesn't define a real worker; therefore, none of the *TEmployee* methods will be called specifically, although they may be inherited. All our employees are either hourly, salaried, or commissioned. The *RunTime* error terminates your program and displays "211", the Call to abstract method error message, if your program mistakenly calls *TEmployee.GetPayAmount*.

Next is the *THourly.GetPayAmount* method considers such things as overtime pay, the number of hours worked, and so on.

```
function THourly.GetPayAmount: Real;
var
  OverTime: Integer;
begin
  Overtime := Time - OvertimeThreshold;
  if Overtime > 0 then
    GetPayAmount := RoundPay(OvertimeThreshold * Rate +
      OverTime * OvertimeFactor * Rate)
  else
    GetPayAmount := RoundPay(Time * Rate);
end;
```

The *TSalaried.GetPayAmount* method is much simpler; it divides the rate of pay by the number of pay periods.

```
function TSalaried.GetPayAmount: Real;
begin
  GetPayAmount := RoundPay(Rate / PayPeriods);
end;
```

If you look at the *TCommissioned.GetPayAmount* method, you'll see it calls *TSalaried.GetPayAmount*, calculates a commission, and adds it to the amount returned by *TSalaried.GetPayAmount*.

```
function TCommissioned.GetPayAmount: Real;
begin
  GetPayAmount := RoundPay(TSalaried.GetPayAmount + Commission *
    SalesAmount);
end;
```

Important!

Whereas methods can be overridden, data fields cannot. Once you define a data field in an object hierarchy, no descendant type can define a data field with precisely the same identifier.

Inheriting static methods

All the methods shown so far in connection with the *TEmployee*, *THourly*, *TSalaried*, and *TCommissioned* object types are static methods. There is a problem inherent with static methods, however.

To understand the problem, let's leave our payroll example, and consider another simplistic and unrealistic, but instructional example. Let's go back to talking about insects. Suppose you want to build a program that will draw different types of flying insects on your screen. You decide to start with a *Winged* object at the top of your hierarchy. You plan to build new flying insect object types as descendants of *Winged*. For example, you might create a *Bee* object type, which differs only from a generic winged insect in that a bee has a stinger and stripes. Of course, a bee has other distinguishing characteristics, but for our example, this is how it might look:

```
type
  Winged = object(Insect)
    procedure Init(AX, AY: Integer)   { initializes an instance }
    procedure Show;       { displays winged insect on the screen }
    procedure Hide;       { erases the winged insect }
    procedure MoveTo(NewX, NewY: Integer);  { moves winged insect }
  end;

type
  Bee = object(Winged)
    .
    .
    procedure Init(AX, AY: Integer) { initializes instance of Bee }
    procedure Show;                 { displays a bee on the screen }
    procedure Hide;                 { erases the bee }
    procedure MoveTo(NewX, NewY: Integer;   { moves the bee }
  end;
```

Both *Winged* and *Bee* have four methods. *Winged.Init* and *Bee.Init* initialize an instance of their respective objects. The *Winged.Show* method knows how to draw a winged insect on the screen; the *Bee.Show* methods knows how to draw a Bee on the screen (a winged insect with stripes and a stinger). The *Winged.Hide* method knows how to erase a winged insect; *Bee.Hide* knows how to erase a bee. The two *Show* methods differ, as do the two *Hide* methods.

The *Winged.MoveTo* and the *Bee.MoveTo* methods are exactly the same, however. In our example, *X* and *Y* define a location on the screen.

```
procedure Winged.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;          { new X coordinate on the screen }
  Y := NewY:          { new Y coordinate on the screen }
  Show;
end;

procedure Bee.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;           { new X coordinate on the screen }
  Y := NewY:           { new Y coordinate on the screen }
  Show;
end;
```

Nothing was changed other than to copy the routine and give it *Bee*'s qualifier in front of the *MoveTo* identifier. Since the methods are identical, why bother to put *MoveTo* into *Bee*? Afterall, *Bee* automatically inherits *MoveTo* from *Winged*. There seems to be no need to override *Winged*'s *MoveTo* method, but this is where the problem with static methods appears.

The term *static* was chosen to describe methods that are not *virtual*. (You will learn about virtual methods shortly.) Virtual methods are in fact the solution to this problem, but in order to understand the solution you must first understand the problem.

The symptoms of the problem are these: Unless a copy of the *MoveTo* method is placed in *Bee*'s scope to override *Winged*'s *MoveTo*, the method does not work correctly when it is called from an object of type *Bee*. If *Bee* invokes *Winged*'s *MoveTo* method, what is moved on the screen is a winged insect rather than a bee. Only when *Bee* calls a copy of the *MoveTo* method defined in its

own scope are bees hidden and drawn by the nested calls to *Show* and *Hide*.

Why so? It has to do with the way the compiler resolves method calls. When the compiler compiles *Bee*'s methods, it first encounters *Winged.Show* and *Winged.Hide* and compiles code for both into the code segment. A little later down the file it encounters *Winged.MoveTo*, which calls both *Winged.Show* and *Winged.Hide*. As with any procedure call, the compiler replaces the source code references to *Winged.Show* and *Winged.Hide* with the addresses of their generated code in the code segment. Thus, when the code for *Winged.MoveTo* is called, it in turn calls the code for *Winged.Show* and *Winged.Hide* and everything's in phase.

So far, this scenario is all classic Turbo Pascal and would have been true (except for the nomenclature) since Turbo Pascal first appeared on the market in 1983. Things change, however, when you get into inheritance. When *Bee* inherits a method from *Winged*, *Bee* uses the method exactly as it was compiled.

Look again at what *Bee* would inherit if it inherited *Winged.MoveTo*:

```
procedure Winged.MoveTo(NewX, NewY: Integer);
begin
  Hide;       { Calls Winged.Hide }
  X := NewX;
  Y := NewY;
  Show;       { Calls Winged.Show }
end;
```

The comments were added to drive home the fact that when *Bee* calls *Winged.MoveTo*, it also calls *Winged.Show* and *Winged.Hide*, not *Bee.Show* and *Bee.Hide*. *Winged.Show* draws a winged insect, not a bee. As long as *Winged.MoveTo* calls *Winged.Show* and *Winged.Hide*, *Winged.MoveTo* can't be inherited. Instead, it must be overridden by a second copy of itself that calls the copies of *Show* and *Hide* defined within its scope; that is, *Bee.Show* and *Bee.Hide*.

The compiler's logic in resolving method calls works like this: When a method is called, the compiler first looks for a method of that name defined within the object type. The *Bee* type defines methods named *Init*, *Show*, *Hide*, and *MoveTo*. If a *Bee* method were to call one of those four methods, the compiler would replace the call with the address of one of *Bee*'s own methods.

If no method by a name is defined within an object type, the compiler goes up to the immediate ancestor type, and looks

within that type for a method of the name called. If a method by that name is found, the address of the ancestor's method replaces the name in the descendant's method's source code. If no method by that name is found, the compiler continues up to the next ancestor, looking for the named method. If the compiler hits the very first (top) object type, it issues an error message indicating that no such method is defined.

But when a static inherited method is found and used, you must remember that the method called is the method exactly as it was defined *and compiled* for the ancestor type. If the ancestor's method calls other methods, the methods called are the ancestor's methods, even if the descendant has methods that override the ancestor's methods.

Virtual methods and polymorphism

The methods discussed so far are static methods. They are static for the same reason that static variables are static: The compiler allocates them and resolves all references to them *at compile time.* As you've seen, objects and static methods can be powerful tools for organizing a program's complexity.

Sometimes, however, they are not the best way to handle methods.

Problems like the one described in the previous section are due to the compile-time resolution of method references. The way out is to be dynamic—and resolve such references at run time. Certain special mechanisms must be in place for this to be possible, but Turbo Pascal provides those mechanisms in its support of virtual methods.

Important!

Virtual methods implement an extremely powerful tool for generalization called polymorphism. *Polymorphism* is Greek for "many shapes," and it is just that: A way of giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

The simplistic hierarchy of winged insects already described provides a good example of polymorphism in action, implemented through virtual methods.

Each object type in our hierarchy represents a different type of figure onscreen: a winged insect or a bee. It certainly makes sense

to say that you can show a point on the screen, or show a circle. Later on, if you were to define objects to represent other types of winged insects such as moths, dragonflies, butterflies, and so on, you could write a method for each that would display that object onscreen. In the new way of object-oriented thinking, you could say that all these insect types types had the ability to show themselves on the screen. That much they all have in common.

What is different for each object type is the *way* it must show itself to the screen. A bee requires stripes be drawn on its body, for example. Any winged insect type can be shown, but the mechanism by which each is shown is specific to each type. One word, "Show," is used to show (literally) many winged insects. Likewise, if we return to our payroll example, the word "GetPayAmount" calculates the amount of pay for several types of employees.

These are examples of what polymorphism is, and virtual methods are how it is done in Turbo Pascal.

Early binding vs. late binding

The difference between a static method call and a virtual method call is the difference between a decision made now and a decision delayed. When you code a static method call, you are in essence telling the compiler, "You know what I want. Go call it." Making a virtual method call, on the other hand, is like telling the compiler, "You don't know what I want—yet. When the time comes, ask the instance."

Think of this metaphor in terms of the *MoveTo* problem mentioned in the previous section. A call to *Bee.MoveTo* can only go to one place: the closest implementation of *MoveTo* up the object hierarchy. In that case, *Bee.MoveTo* would still call *Winged*'s definition of *MoveTo*, since *Winged* is the closest up the hierarchy from *Bee*. Assuming that no descendant type defined its own *MoveTo* to override *Winged*'s *MoveTo*, any descendant type of *Winged* would still call the same implementation of *MoveTo*. The decision can be made at compile time and that's all that needs to be done.

When *MoveTo* calls *Show*, however, it's a different story. Every figure type has its own implementation of *Show*, so which implementation of *Show* is called by *MoveTo* should depend entirely on what object instance originally called *MoveTo*. This is why the call to the *Show* method within the implementation of *MoveTo* must be

a delayed decision: When the code for *MoveTo* is compiled, no decision as to which *Show* to call can be made. The information isn't available at compile time, so the decision has to be deferred until run time, when the object instance calling *MoveTo* can be queried.

The process by which static method calls are resolved unambiguously to a single method by the compiler at compile time is *early binding*. In early binding, the caller and the callee are connected (bound) at the earliest opportunity, that is, at compile time. With *late binding*, the caller and the callee cannot be bound at compile time, so a mechanism is put into place to bind the two later on, when the call is actually made.

The nature of the mechanism is interesting and subtle, and you'll see how it works a little later.

Object type compatibility

Inheritance somewhat changes Turbo Pascal's type compatibility rules. In addition to everything else, a descendant type inherits type compatibility with all its ancestor types. This extended type compatibility takes three forms:

- between object instances
- between pointers to object instances
- between formal and actual parameters

In all three forms, however, it is critical to remember that type compatibility extends *only* from descendant to ancestor. In other words, descendant types can be freely used in place of ancestor types, but not vice versa.

In WORKERS.TPU, *TSalaried* is a descendant of *TEmployee*, and *TCommissioned* is a descendant of *TSalaried*. With this in mind, consider these declarations:

```
type
  PEmployee = ^TEmployee;
  PSalaried = ^TSalaried;
  PCommissioned = ^TCommissioned;
```

```
var
  AnEmployee: TEmployee;
  ASalaried: TSalaried;
  ACommissioned: TCommissioned;
  TEmployeePtr: PEmployee;
  TSalariedPtr: PSalaried;
  TCommissionedPtr: PCommissioned;
```

With these declarations, the following assignments are legal:

An ancestor object can be assigned an instance of any of its descendant types.

```
AnEmployee := ASalaried;
ASalaried := ACommissioned;
AnEmployee := ACommissioned;
```

The reverse assignments are not legal.

This is a concept new to Pascal, and it might be a little hard to remember, at first, which way the type compatibility goes. Think of it this way: *The source must be able to completely fill the destination.* Descendant types contain everything their ancestor types contain by virtue of inheritance. Therefore a descendant type is either exactly the same size or (usually) larger than its ancestors, but never smaller. Assigning an ancestor object to a descendant object could leave some of the descendant's fields undefined after the assignment, which is dangerous and therefore illegal.

In an assignment statement, only the fields that the two types have in common are copied from the source to the destination. In the assignment statement

```
AnEmployee := ACommissioned;
```

only the *Name, Title,* and *Rate* fields of *ACommissioned* are copied to *AnEmployee,* since *Name, Title,* and *Rate* are all that types *TCommissioned* and *TEmployee* have in common.

Type compatibility also operates between pointers to object types, under the same rule as for instances of object types: Pointers to descendants can be assigned to pointers to ancestors. These pointer assignments are also legal:

```
TSalariedPtr := TCommissionedPtr;
TEmployeePtr := TSalariedPtr;
TEmployeePtr := TCommissionedPtr;
```

Again, the reverse assignments are not legal.

A formal parameter (either value or **var**) of a given object type can take as an actual parameter an object of its own, or any descendant type. Given this procedure header,

```
procedure CalcFedTax(Victim: TSalaried);
```

actual parameters could legally be of type *TSalaried* or *TCommissioned*, but not type *TEmployee*. *Victim* could also be a **var** parameter; the same type compatibility rule applies.

Warning!

However, keep in mind that there's a drastic difference between a value parameter and a **var** parameter: A **var** parameter is a pointer to the actual object passed as a parameter, whereas a value parameter is only a *copy* of the actual parameter. That copy, moreover, only includes the fields and methods included in the formal value parameter's type. This means the actual parameter is literally translated to the type of the formal parameter. A **var** parameter is more similar to a typecast, in that the actual parameter remains unaltered.

Similarly, if a formal parameter is a pointer to an object type, the actual parameter can be a pointer to that object type or a pointer to any of that object's descendant types. Given this procedure header,

```
procedure Worker.Add(AWorker: PSalaried);
```

actual parameters could legally be of type *PSalaried* or *PCommissioned*, but not type *PEmployee*.

Polymorphic objects

In reading the previous section, you might have asked yourself: If any descendant type of a parameter's type can be passed in the parameter, how does the user of the parameter know which object type it is receiving? In fact, the user does not know, not directly. The exact type of the actual parameter is unknown at compile time. It could be any one of the object types descended from the **var** parameter type and is thus called a *polymorphic object*.

Now, exactly what are polymorphic objects good for? Primarily, this: *Polymorphic objects allow the processing of objects whose type is not known at compile time.* This whole notion is so new to the Pascal way of thinking that an example might not occur to you immediately. (You'll be surprised, in time, at how natural it begins to seem.)

Suppose you've written a toolbox that draws numerous types of winged insects: butterflies, bees, moths, and so on. You want to write a routine that drags insects around the screen with the mouse pointer.

The old way would have been to write a separate drag procedure for each type of insect. You would have had to write *DragButterfly*, *DragBee*, *DragMoth*, and so on. Even if the strong typing of Pascal allowed it (and don't forget, there are always ways to circumvent strong typing), the differences between the types of insects would seem to prevent a truly general dragging routine from being written.

After all, a bee has stripes and a stinger, a butterfly has large, colorful wings, a dragonfly has iridescent colors, arrgh....

At this point, clever Turbo Pascal hackers will step forth and say, do it this way: Pass the winged insect record to procedure *DragIt* as the referent of a generic pointer. Inside *DragIt*, examine a tag field at a fixed offset inside the winged insect record to determine what sort of insect it is, and then branch using a **case** statement:

```
case FigureIDTag of
  Bee          : DragBee;
  Butterfly    : DragButterfly;
  Dragonfly    : DragDragonfly;
  Mosquito     : DragMosquito;
  ...
```

Well, placing seventeen small suitcases inside one enormous suitcase is a slight step forward, but what's the real problem with this way of doing things?

What if the user of your toolbox defines some new winged insect type?

What indeed? What if the user wants to work with Mediterranean fruitflies? Your program does not have a Fruitfly type, so *DragIt* would not have a Fruitfly label in its **case** statement, and would therefore refuse to drag the new Fruitfly figure. If it were presented to *DragIt*, Fruitfly would fall out in the **case** statement's **else** clause as an "unrecognized insect."

Plainly, building a toolbox of routines for sale without source code suffers from this problem: The toolbox can only work on data types that it "knows," that is, that are defined by the designers of the toolbox. The user of the toolbox is powerless to extend the function of the toolbox in directions unanticipated by the toolbox designers. What the user buys is what the user gets. Period.

The way out is to use Turbo Pascal's extended type compatibility rules for objects and design your application to use polymorphic

objects and virtual methods. If a toolbox *DragIt* procedure is set up to work with polymorphic objects, it works with any objects defined within the toolbox—and any descendant objects that you define yourself. If the toolbox object types use virtual methods, the toolbox objects and routines can work with your custom winged insects figures *on the figures' own terms*. A virtual method you define today is callable by a toolbox .TPU unit file that was written and compiled a year ago. Object-oriented programming makes it possible, and virtual methods are the key.

Understanding how virtual methods make such polymorphic method calls possible requires a little background on how virtual methods are declared and used.

Virtual methods

A method is made virtual by following its declaration in the object type with the new reserved word **virtual**. Remember that if you declare a method in an ancestor type **virtual**, all methods of the same name in any descendant must also be declared **virtual** to avoid a compiler error.

Here are the employee objects you have seen in the previous payroll example, properly virtualized:

```
type
  PEmployee = ^TEmployee;
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate: Real;
    constructor Init(AName, ATitle: String; ARate: Real);
    function GetPayAmount: Real; virtual;
    function GetName: String;
    function GetTitle: String;
    function GetRate: Real;
    procedure Show; virtual;
  end;

  PHourly = ^THourly;
  THourly = object(TEmployee)
    Time: Integer;
    constructor Init(AName, ATitle: String; ARate: Real; ATime:
      Integer);
    function GetPayAmount: Real; virtual;
    function GetTime: Integer;
  end;
```

```
PSalaried = ^TSalaried;
TSalaried = object(TEmployee)
  function GetPayAmount: Real; virtual;
end;

PCommissioned = ^TCommissioned;
TCommissioned = object(TSalaried)
  Commission: Real;
  SalesAmount: Real;
  constructor Init(AName, ATitle: String;
    ARate, ACommission, ASalesAmount: Real);
  function GetPayAmount: Real; virtual;
end;
```

And here is the insect example, complete with virtual methods:

```
type
  Winged = object(Insect)
    constructor Init(AX, AY: Integer)
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo(NewX, NewY: Integer;
  end;

type
  Bee=object(Winged)
    constructor Init(AX, AY: Integer)
    procedure Show; virtual;
    procedure Hide; virtual;
  end;
```

Notice first of all that the *MoveTo* method shown of type *Bee* is gone from *Bee's* type definition. *Bee* no longer needs to override *Winged's MoveTo* method with an unmodified copy compiled within its own scope. Instead, *MoveTo* can now be inherited from *Winged*, with all *MoveTo's* nested method calls going to *Bee's* methods rather than *Winged's*, as happens in an all-static object hierarchy.

We suggest the use of the identifier Init for object constructors.

Also, notice the new reserved word **constructor** replacing the reserved word **procedure** for *Winged.Init* and *Bee.Init*. A constructor is a special type of procedure that does some of the setup work for the machinery of virtual methods.

Warning! *Every object type that has virtual methods must have a constructor.*

The constructor must be called before any virtual method is called. Calling a virtual method without previously calling the constructor can cause system lockup, and the compiler has no way to check the order in which methods are called.

Each individual instance of an object must be initialized by a separate constructor call. It is not sufficient to initialize one instance of an object and then assign that instance to additional instances. The additional instances, while they might contain correct data, are not initialized by the assignment statements, and lock up the system if their virtual methods are called. For example

```
var
  FBee, GBee: Bee;                    { create two instances of Bee }

begin
  FBee.Init(5, 9);                     { call constructor for FBee }
  GBee := FBee;                              { GBee is not valid! }
end.
```

What do constructors construct? Every object type has something called a *virtual method table* (VMT) in the data segment. The VMT contains the object type's size and, for each of its virtual methods, a pointer to the code implementing that method. What the constructor does is establish a link between the instance calling the constructor and the object type's VMT.

That's important to remember: There is only one virtual method table for each object type. Individual instances of an object type (that is, variables of that type) contain a link to the VMT—they do not contain the VMT itself. The constructor sets the value of that link to the VMT—which is why you can launch execution into nowhere by calling a virtual method before calling the constructor.

Range checking virtual method calls

*The default state of **$R** is inactive, {**$R-**}.*

During program development, you might wish to take advantage of a safety net that Turbo Pascal places beneath virtual method calls. If the **$R** toggle is in its active state, {**$R+**}, all virtual method calls are checked for the initialization status of the instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs.

Once you've shaken out a program and are certain that no method calls from uninitialized instances are present, you can speed your code up somewhat by setting the **$R** toggle to its inactive state, {**$R-**}. Method calls from uninitialized instances will no longer be checked for, and will probably lock up your system if they're found.

Once virtual, always virtual

Notice that both *Winged* and *Bee* have methods named *Show* and *Hide*. All method headers for *Show* and *Hide* are tagged as virtual methods with the reserved word **virtual**. Once an ancestor object type tags a method as **virtual**, all its descendant types that implement a method of that name must tag that method **virtual** as well. In other words, a static method can never override a virtual method. If you try, a compiler error results.

You should also keep in mind that the method heading cannot change in *any* way downward in an object hierarchy once the method is made virtual. You might think of each definition of a virtual method as a gateway to *all* of them. For this reason, the headers for all implementations of the same virtual method must be identical, right down to the number and type of parameters. This is not the case for static methods; a static method overriding another can have different numbers and types of parameters as necessary.

It's a whole new world.

Object extensibility

The important thing to notice about units like WORKERS.PAS is that the object types and methods defined in the unit can be distributed to users in linkable .TPU form only, without source code. (Only a listing of the interface portion of the unit need be released.) Using polymorphic objects and virtual methods, the users of the .TPU file can still add features to it to suit their needs.

This novel notion of taking someone else's program code and adding functionality to it *without benefit of source code* is called *extensibility*. Extensibility is a natural outgrowth of inheritance: You inherit everything that all your ancestor types have, and then you add what new capability you need. Late binding lets the new meld with the old at run time, so the extension of the existing code is seamless and costs you no more in performance than a quick trip through the virtual method table.

Static or virtual methods

In general, you should make methods virtual. Use static methods only when you want to optimize for speed and memory efficiency. The tradeoff, as you've seen, is in extensibility.

Let's say you are declaring an object named *Ancestor*, and within *Ancestor* you are declaring a method named *Action*. How do you decide whether *Action* should be virtual or static? Here's the rule of thumb: Make *Action* virtual if there is a possibility that some future descendant of *Ancestor* will override *Action*, and you want that future code to be accessible to *Ancestor*.

On the other hand, remember that if an object has any virtual methods, a VMT is created for that object type in the data segment and every object instance has a link to the VMT. Every call to a virtual method must pass through the VMT, while static methods are called directly. Though the VMT lookup is very efficient, calling a method that is static is still a little faster than calling a virtual one. And if there are no virtual methods in your object, then there is no VMT in the data segment and—more significantly—no link to the VMT in every object instance.

The added speed and memory efficiency of static methods must be balanced against the flexibility that virtual methods allow: extension of existing code long after that code is compiled. Keep in mind that users of your object type might think of ways to use it that you never dreamed of, which is, after all, the whole point.

Dynamic objects

All the object examples shown so far have had static instances of object types that were named in a **var** declaration and allocated in the data segment and on the stack.

The use of the word static here does not relate in any way to static methods.

```
var
  ASalaried: TSalaried;
```

Objects can be allocated on the heap and manipulated with pointers, just as the closely related record types have always been in Pascal. Turbo Pascal includes some powerful extensions to make dynamic allocation and deallocation of objects easier and more efficient.

Objects can be allocated as pointer referents with the *New* procedure:

```
var
  CurrentPay: Real;
  P: ^TSalaried;

New(P);
```

As with record types, *New* allocates enough space on the heap to contain an instance of the pointer's base type, and returns the address of that space in the pointer.

If the dynamic object contains virtual methods, it must then be initialized with a constructor call before any calls are made to its methods:

```
P^.Init('Sara Adams', 'Account manager', 2400);
```

Method calls can then be made normally, using the pointer name and the reference symbol ^ (a caret) in place of the instance name that would be used in a call to a statically allocated object:

```
CurrentPay := P^.GetPayAmount;
```

Allocation and initialization with New

Turbo Pascal extends the syntax of *New* to allow a more compact and convenient means of allocating space for an object on the heap and initializing the object with one operation. *New* can now be invoked with two parameters: the pointer name as the first parameter, and the constructor invocation as the second parameter:

```
New(P, Init('Sara Adams', 'Account manager', 2400));
```

When you use this extended syntax for *New*, the constructor *Init* actually performs the dynamic allocation, using special entry code generated as part of a constructor's compilation. The instance name cannot precede *Init*, since at the time *New* is called, the instance being initialized with *Init* does not yet exist. The compiler identifies the correct *Init* method to call through the type of the pointer passed as the first parameter.

New has also been extended to allow it to act as a function returning a pointer value. The parameter passed to *New* is the *type* of the pointer to the object rather than the pointer variable itself:

```
type
  PSalaried = ^TSalaried;

var
  P: PSalaried;

P := New(PSalaried);
```

Note that with this version, the function-form extension to *New* applies to *all* data types, not only to object types.

The function form of *New*, like the procedure form, can also take the object type's constructor as a second parameter:

```
P := New(PSalaried, Init('Sara Adams', 'Account manager',
  2400));
```

Fail helps you do error recovery in constructors; see the section "Constructor error recovery" in Chapter 17 of the Programmer's Guide.

A parallel extension to *Dispose* has been defined for Turbo Pascal, as fully explained in the following sections.

Disposing dynamic objects

Just like traditional Pascal records, objects allocated on the heap can be deallocated with *Dispose* when they are no longer needed:

```
Dispose(P);
```

There can be more to getting rid of an unneeded dynamic object than just releasing its heap space, however. An object can contain pointers to dynamic structures or objects that need to be released or "cleaned up" in a particular order, especially when elaborate dynamic data structures are involved. Whatever needs to be done to clean up a dynamic object in an orderly fashion should be gathered together in a single method so that the object can be eliminated with one method call:

```
MyComplexObject.Done;
```

We suggest the identifier Done for cleanup methods that "close up shop" once an object is no longer needed.

The *Done* method should encapsulate all the details of cleaning up its object and all the data structures and objects nested within it.

It is legal and often useful to define multiple cleanup methods for a given object type. Complex objects might need to be cleaned up in different ways depending on how they were allocated or used, or depending on what mode or state the object was in when it was cleaned up.

Destructors

Turbo Pascal provides a special type of method called a *destructor* for cleaning up and disposing of dynamically allocated objects. A destructor combines the heap deallocation step with whatever other tasks are necessary for a given object type. As with any method, multiple destructors can be defined for a single object type.

A destructor is defined with all the object's other methods in the object type definition:

```
type
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate: Real;
    constructor Init(AName, ATitle: String; ARate: Real);
    destructor Done; virtual;
    function GetName: String;
    function GetTitle: String;
    function GetRate: Rate; virtual;
    function GetPayAmount: Real; virtual;
  end;
```

Destructors can be inherited, and they can be either static or virtual. Because different shutdown tasks are usually required for different object types, it is a good idea *always* to make destructors virtual, so that in every case the correct destructor is executed for its object type.

Keep in mind that the reserved word **destructor** is not needed for every cleanup method, even if the object type definition contains virtual methods. Destructors really operate only on dynamically allocated objects. In cleaning up a dynamically allocated object, the destructor performs a special service: It guarantees that the correct number of bytes of heap memory are always released. There is, however, no harm in using destructors with statically allocated objects; in fact, by not giving an object type a destructor, you prevent objects of that type from getting the full benefit of Turbo Pascal's dynamic memory management.

Destructors really come into their own when polymorphic objects must be cleaned up and their heap allocation released. A polymorphic object is an object that has been assigned to an ancestor type by virtue of Turbo Pascal's extended type compatibility rules. An instance of object type *THourly* assigned to a variable of type *TEmployee* is an example of a polymorphic object. These rules govern pointers to objects as well; a pointer to *THourly* can be freely assigned to a pointer to type *TEmployee*, and the referent of that pointer is also a polymorphic object.

The term *polymorphic* is appropriate because the code using the object doesn't know at compile time precisely what type of object is on the end of the string—only that the object is one of a hierarchy of objects descended from the specified type.

The size of object types differ, obviously. So when it comes time to clean up a polymorphic object allocated on the heap, how does *Dispose* know how many bytes of heap space to release? No information on the size of the object can be gleaned from a polymorphic object at compile time.

The destructor solves the problem by going to the place where the information *is* stored: in the instance variable's VMT. In every object type's VMT is the size in bytes of the object type. The VMT for any object is available through the invisible *Self* parameter passed to the method on any method call. A destructor is just a special kind of method, and it receives a copy of *Self* on the stack when an object calls it. So while an object might be polymorphic at *compile time,* it is never polymorphic at run time, thanks to late binding.

To perform this late-bound memory deallocation, the destructor must be called as part of the extended syntax for the *Dispose* procedure:

```
Dispose(P, Done);
```

(Calling a destructor outside of a *Dispose* call does no automatic deallocation at all.) What happens here is that the destructor of the object pointed to by *P* is executed as a normal method call. As the last thing it does, however, the destructor looks up the size of its instance type in the instance's VMT, and passes the size to *Dispose*. *Dispose* completes the shutdown by deallocating the correct number of bytes of heap space that had previously belonged to *P^*. The number of bytes released is correct whether *P* points to an instance of type *TSalaried* or to one of *TSalaried*'s descendant types like *TCommissioned*.

Note that the destructor method itself can be empty and still perform this service:

```
destructor AnObject.Done;
begin
end;
```

What performs the useful work in this destructor is not the method body but the epilog code generated by the compiler in response to the reserved word **destructor**. In this, it is similar to a unit that exports nothing, but performs some "invisible" service by executing an initialization section before program startup. The action is all behind the scenes.

An example of dynamic object allocation

The final example program provides some practice in the use of objects allocated on the heap, including the use of destructors for object deallocation. The program shows how a linked list of worker objects might be created on the heap and cleaned up using destructor calls when they are no longer required.

Building a linked list of objects requires that each object contain a pointer to the next object in the list. Type *TEmployee* contains no such pointer. The easy way out would be to add a pointer to *TEmployee*, and in doing so ensure that all *TEmployee*'s descendant types also inherit the pointer. However, adding anything to *TEmployee* requires that you have the source code for *TEmployee*, and as said earlier, one advantage of object-oriented programming is the ability to extend existing objects without necessarily being able to recompile them.

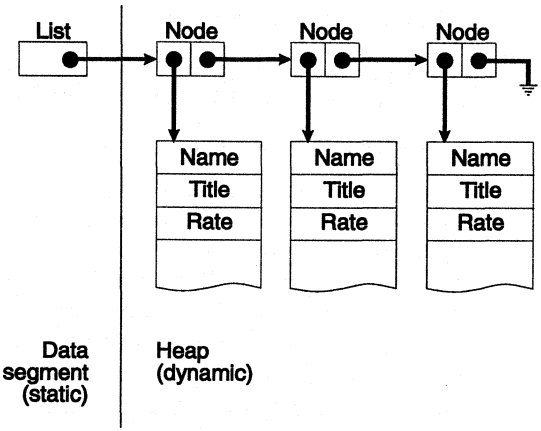
The solution that requires no changes to *TEmployee* creates a new object type not descended from *TEmployee*. Type *TStaffList* is a very simple object whose purpose is to head up a list of *TEmployee* objects. Because *TEmployee* contains no pointer to the next object in the list, a simple record type, *TNode*, provides that service. *TNode* is even simpler than *TStaffList*, in that it is not an object, has no methods, and contains no data except a pointer to type *TEmployee* and a pointer to the next node in the list.

TStaffList has a method that allows it to add new workers to its linked list of *TNode* records by inserting a new instance of *TNode* immediately after itself, as a referent to its TNodes pointer field. The *Add* method takes a pointer to an *TEmployee* object, rather than an *TEmployee* object itself. Because of Turbo Pascal's extended type compatibility, pointers to any type descended from *TEmployee* can also be passed in the *Item* parameter to *TStaffList.Add*.

Program *WorkList* declares a static variable, *Staff*, of type *TStaffList*, and builds a linked list with five nodes. Each node points to a worker object that is one of *TEmployee*'s descendants. The number of bytes of free heap space is reported before any of the dynamic objects are created, and then again after all have been created. Finally, the whole structure, including the five *TNode* records and the five *TEmployee* objects, are cleaned up and removed from the

heap with a single destructor call to the static *TStaffList* object, *Staff*.

Figure 4.2
Layout of program WorkList's data structures



Disposing of a complex data structure on the heap

This destructor, *Staff.Done*, is worth a close look. Shutting down a *TStaffList* object involves disposing of three different kinds of structures: the polymorphic worker objects in the list, the *TNode* records that hold the list together, and (if it is allocated on the heap) the *TStaffList* object that heads up the list. The whole process is invoked by a single call to *TStaffList*'s destructor:

```
Staff.Done;
```

The code for the destructor merits examination:

```
destructor TStaffList.Done;
var
  N: TNodePtr;
begin
  while TNodes <> nil do
  begin
    N := TNodes;
    Dispose(N^.Item, Done);
    TNodes := N^.Next;
    Dispose(N);
  end;
end;
```

The list is cleaned up from the list head by the "hand-over-hand" algorithm, metaphorically similar to pulling in the string of a kite: Two pointers, the *TNodes* pointer within *Staff* and a working pointer *N*, alternate their grasp on the list while the first item in the list is disposed of. A dispose call deallocates storage for the

first *TEmployee* object in the list (*Item*^); then *TNodes* is advanced to the next *TNode* record in the list by the statement `TNodes := N^.Next`; the *TNode* record itself is deallocated; and the process repeats until the list is gone.

The important thing to note in the destructor *Done* is the way the *TEmployee* objects in the list are deallocated:

```
Dispose(N^.Item,Done);
```

Here, *N*^*.Item* is the first *TEmployee* object in the list, and the *Done* method called is its destructor. Keep in mind that the actual type of *N*^*.Item*^ is not necessarily *TEmployee*, but could as well be any descendant type of *TEmployee*. The object being cleaned up is a polymorphic object, and no assumptions can be made about its actual size or exact type at compile time. In the earlier call to *Dispose*, once *Done* has executed all the statements it contains, the "invisible" epilog code in *Done* looks up the size of the object instance being cleaned up in the object's VMT. *Done* passes that size to *Dispose*, which then releases the exact amount of heap space the polymorphic object actually occupied.

Remember that polymorphic objects must be cleaned up this way, through a destructor call passed to *Dispose*, if the correct amount of heap space is to be reliably released.

In the example program, *Staff* is declared as a static variable in the data segment. *Staff* could as easily have been itself allocated on the heap, and anchored to reality by a pointer of type *TStaffListPtr*. If the head of the list had been a dynamic object too, disposing of the structure would have been done by a destructor call executed within *Dispose*:

```
var
  Staff: TStaffListPtr;
...
Dispose(Staff, Done);
```

Here, *Dispose* calls the destructor method *Done* to clean up the structure on the heap. Then, once *Done* is finished, *Dispose* deallocates storage for *Staff*'s referent, removing the head of the list from the heap as well.

WORKLIST.PAS (on your disk) uses the same WORKERS.PAS unit described on page 100. It creates a *List* object heading up a linked list of five polymorphic objects compatible with *TEmployee*, produces a payroll report, and then disposes of the whole dynamic data structure with a single destructor call to *Staff.Done*.

Where to now?

As with any aspect of computer programming, you don't get better at object-oriented programming by reading about it; you get better at it by doing it. Most people, on first exposure to object-oriented programming, are heard to mutter "I don't get it" under their breath. The "Aha!" comes later, when in the midst of putting their own objects in place, the whole concept comes together in the sort of perfect moment we used to call an epiphany. Like the face of woman emerging from a Rorschach inkblot, what was obscure before at once becomes obvious, and from then on it's easy.

The best thing to do for your first object-oriented project is to take the WORKERS.PAS unit (you have it on disk) and extend it. Once you've had your "Aha!," start building object-oriented concepts into your everyday programming chores. Take some existing utilities you use every day and rethink them in object-oriented terms. Take another look at your hodgepodge of procedure libraries and try to see the objects in them—then rewrite the procedures in object form. You'll find that libraries of objects are much easier to reuse in future projects. Very little of your initial investment in programming effort will ever be wasted. You will rarely have to rewrite an object from scratch. If it will serve as is, use it. If it lacks something, extend it. But if it works well, there's no reason to throw away any of what's there.

Conclusion

Object-oriented programming is a direct response to the complexity of modern applications, complexity that has often made many programmers throw up their hands in despair. Inheritance and encapsulation are extremely effective means for managing complexity. (It's the difference between having ten thousand insects classified in a taxonomy chart, and ten thousand insects all buzzing around your ears.) Far more than structured programming, object-orientation imposes a rational order on software structures that, like a taxonomy chart, imposes order without imposing limits.

Add to that the promise of the extensibility and reusability of existing code, and the whole thing begins to sound almost too good to be true. Impossible, you think?

Hey, this is Turbo Pascal. "Impossible" is undefined.

C H A P T E R

5

Project management

So far, you've learned how to write Turbo Pascal programs, how to use the predefined units, and how to write your own units. At this point, your program could become quite large, perhaps separated into multiple source files. How do you manage such a programming project?

This chapter tells you how to

- manage a programming project within the IDE
- organize your program into units
- take advantage of the built-in Make and Build options
- use conditional compilation within a source code file
- optimize your code for speed

Working in the IDE

While writing and editing your programs, you will probably set options in the Preferences dialog box, turn on or off compiler and linker options, or resize and rearrange your edit windows to your liking. You can tell Turbo Pascal to remember what you have done so the next time you start a programming session, your work environment will appear just as you left it—including having the same files open as when you exited. This next section will help you understand how this works.

Saving your working environment

Turbo Pascal remembers the settings and files you used during a programming session and uses them the next time you start another session.

If you haven't turned off the Options | Preferences | Auto Save options, Desktop and Configuration, when you exit Turbo Pascal your settings are saved in a configuration file and a desktop file. The next time you start Turbo Pascal, everything will be just as you left it.

The configuration file

The configuration file is the key to Turbo Pascal's memory. The first time you start up Turbo Pascal, a TPW.CFG file is created. If you have Auto Save on for the configuration file, each time you exit Turbo Pascal, the configuration file is updated. All the options you set with the Options menu are saved as well as the settings you made in the Find Text dialog box and the name of the primary file.

The desktop file

With the Desktop Auto Save option on (the default setting) in the Options | Preferences dialog box, a desktop file is updated each time you exit Turbo Pascal. A desktop file keeps track of the files you had open as well as files you worked on but closed during a programming session (except NONAMExx.PAS files). When you start up a new session, your edit windows will appear just as you left them. When you open the File menu, you will see a list of closed files that you opened sometime in the past. As long as you keep using the same desktop file, the list of closed files on the File menu will continue to grow to a maximum of five files.

So how do you determine which desktop file to use? You can't select a new desktop file directly, but every time you create a new configuration file, Turbo Pascal will create a new desktop file. The file name will be the same except a desktop file will have a .DSK extension instead of a .CFG extension. For example, if your configuration file is named MY.CFG, then the desktop file will be MY.DSK.

Clearing your desktop

At times you may want to keep your current configuration settings, but clear your desktop so that Turbo Pascal "forgets" the list of files you have been working with. You can do this by choosing Options | Open to bring up the Open Configuration File dialog box and typing the name of the *current* configuration file in the input box. Choose *Enter*. Turbo Pascal remembers your current settings, but your desktop is cleared.

Managing an IDE programming project

If you work on a project that uses several unit or Include files, you'll want to specify a primary file, the file that gets compiled when you use Compile | Make or Build. Use the Compile | Primary File to specify a primary file.

Each project you work on in the IDE has unique requirements. For example, each may have a different primary file, if you specify one. You can customize the IDE to your project by using the Desktop and Configuration Auto Save options. When you begin a new project, create a new configuration file. Set all options the way you want them to be for the new project and then choose Save As from the Options menu. A dialog box prompts you for a new configuration file name. Type in a new name and choose OK.

If you exit Turbo Pascal at this point and the Desktop and Configuration Auto Save options are on, the next time you start a new session, Turbo Pascal will use the new configuration file. The files you were working with will be readily available to you, either in an edit window or in the closed files listing on the File menu, because a new desktop file has also been created for your project.

If you want to work on a different project, you can load another configuration file or create a new one using Open on the Options menu. The secret to project management in the IDE is using a different configuration file for each project.

How does Turbo Pascal know which configuration file to use when you start a new session? The answer is in the TPW.INI file. You'll find TPW.INI in your Windows directory. If you open it in an edit window, you'll see that it contains the path and name of your current configuration file as well as a series of numbers. The numbers indicate the size and location of the Turbo Pascal

desktop window when Turbo Pascal starts up. This file is updated each time you exit. If you delete the TPW.INI file, Turbo Pascal won't know which configuration file to use, so it will use the default: TPW.CFG. If there is no such file, it will create one.

You can also use the TPW.INI file to specify command-line options when running Turbo Debugger for Windows. Open TPW.INI in an edit window and add this line at the bottom of the file:

```
[Debugger]
```

You can specify the directory where Turbo Pascal looks for Turbo Debugger for Windows and you can specify the command-line options you want to use when running Turbo Debugger.

To specify a directory, type in this line immediately after [Debugger],

```
Exepath=
```

and type in the full path name of the directory where you keep the debugger.

To specify command-line options, type in this line,

```
Switches=
```

and include the command-line options you want Turbo Debugger to use.

For example,

```
[Debugger]
Exepath=C:\BIN
Switches=-cMYCONFIG.TDW
```

For a complete list of Turbo Debugger command-line options, see the Turbo Debugger for Windows User's Guide.

will search for Turbo Debugger for Windows in the C:\BIN directory and will use the settings in the MYCONFIG.TDW configuration file.

There is one more subject we should discuss before we leave the topic of managing an IDE project: the current work directory.

Where are my files?

If this is your first attempt at programming under Windows, you might be confused as to where Turbo Pascal looks for and saves your files. Turbo Pascal uses the *current work directory*. The following list explains how the current work directory is determined.

- The directory that holds the primary file, if you specified one, becomes the current work directory. You can include a full path name when naming a primary file.
- If you didn't specify a primary file, the directory that contains the file in the active edit window becomes the current work directory. You can include a full path name when saving a file in an edit window.
- If there is no active edit window, the directory TPW.EXE is in becomes the current work directory.

Once a file is open or created, Turbo Pascal remembers its full path even if the current work directory happens to change.

Since the current work directory is usually determined either by the primary file or the file in the active edit window, and these items are saved in the configuration and desktop files, the configuration and desktop files indirectly determine the current work directory for a project.

Working with files in another directory

To open a file in another directory, choose File | Open and simply type in the complete path and file name in the input box. Or you can use the directory list to display files in another directory and select the file you want. Once you specify a file name and choose *Enter*, the next time you choose File | Open in the same session, you will see the files in this other directory. Your current work directory has not changed, however. If you create and save a new file, Turbo Pascal will save it in the current work directory.

If you want to work with files in more than one directory, don't forget the convenience of the history list in the File Open dialog box. Click the drop-down arrow to the right of the input box or press *Alt+* ↓ to see the history list. Chances are the files you want may already be part of the history list. Or you can type a path and file-name specification including wildcards for the directory that contains the files you want. For example,

```
C:\OTHER\*.PAS
```

displays the Pascal source files in the C:\OTHERDIR directory. The next time you want to see the files in this directory, choose C:\OTHERDIR from the history list.

Program organization

Turbo Pascal allows you to divide your program into code segments. Your main program is a single code segment, which means that after compilation, it can have no more than 64K of machine code. You can exceed this limit, however, by breaking your program up into units. Each unit can also contain up to 64K of machine code when compiled. The question is: How should you organize your program into units?

The first step is to collect all your global definitions—constants, data types, and variables—into a single unit; let's call it *MyGlobals*. This is necessary if your other units reference those definitions. Unlike Include files, units can't "see" any definitions made in your main program; they can only see what's in the interface section of their own unit and other units they use. Your units can use *MyGlobals* and thus reference all your global declarations.

A second possible unit is *MyUtils*. In this unit you could collect all the utility routines used by the rest of your program. These would have to be routines that don't depend on any others (except possibly other routines in *MyUtils*).

Beyond that, you should collect procedures and functions into logical groups. In each group, you'll often find a few procedures and functions that are called by the rest of the program, and then several (or many) procedures/functions that are called by those few. A group like that makes a wonderful unit. Here's how to convert it:

1. Copy all those procedures and functions into a separate file and delete them from your main program.
2. Open that file for editing.
3. Type the following lines in front of those procedures and functions:

   ```
   unit unitname;
   interface
   uses MyGlobals;
   implementation
   ```

   where *unitname* is the name of your unit (and also the name of the file you're editing).
4. Type **end.** at the very end of the file.

5. In the space between **interface** and **implementation**, copy the procedure and function headers of those routines called by the rest of the program. Those headers are simply the first line of each routine, the one that starts with **procedure** (or **function**).
6. If this unit needs to use any others, type their names (separated by commas) between *MyGlobals* and the semicolon in the **uses** statement.
7. Compile the unit you've created.
8. Go back to your main program and add the unit's name to the **uses** statement at the start of the program.

Ideally, you want your program organized so that when you are working on a particular aspect of it, you are modifying and recompiling a single module (unit or main program). This minimizes compile time; more importantly, it lets you work with smaller, more manageable chunks of code.

Initialization

Remember in all this that each unit can (optionally) have its own initialization code. This code is automatically executed when the program is first loaded. If your program uses several units, the initialization code for each unit is executed. The order of execution follows the order in which the units are listed in your program's **uses** statement; so if your program has the statement

```
uses MyGlobals,MyUtils,EditLib,GraphLib;
```

then the initialization section (if any) of *MyGlobals* will be called first, followed by that of *MyUtils*, then *EditLib*, then *GraphLib*.

To create an initialization section for a unit, put the keyword **begin** above the **end** that ends the implementation section. This defines the initialization section of your unit, much as the **begin..end** pair defines the main body of a program, a procedure, or a function. You can then put any Pascal code you want in here. It can reference everything declared in that unit, in both the public (interface) and private (implementation) sections; it can also reference anything from the interface portions of any units that this unit uses.

Turbo Pascal has an important feature to aid you in project management: a built-in Make utility. To understand its significance, let's look at the previous example again.

Suppose you have a program, MYAPP.PAS, which uses four units: *MyGlobals*, *MyUtils*, *EditLib*, and *GraphLib*. Those four units are contained in the text files MYGLOBAL.PAS, MYUTILS.PAS, EDITLIB.PAS, and GRAPHLIB.PAS, respectively. Furthermore, *MyUtils* uses *MyGlobals*, and *EditLib* and *GraphLib* use both *MyGlobals* and *MyUtils*.

When you compile MYAPP.PAS, it looks for the files MYGLOBAL.TPU, MYUTILS.TPU, EDITLIB.TPU, and GRAPHLIB.TPU, loads them into memory, links them with the code produced by compiling MYAPP.PAS, and writes everything out to MYAPP.EXE. So far, so good.

Suppose now you make modifications to EDITLIB.PAS. In order to recreate MYAPP.EXE, you need to recompile both EDITLIB.PAS and MYAPP.PAS. A little tedious, but no problem.

Now, suppose you modify the interface section of MYGLOBAL.PAS. To update MYAPP.EXE, you have to recompile all four units, as well as MYAPP.PAS. That means five separate compilations each time you make a change to MYGLOBAL.PAS—which could be enough to discourage you from using units at all. But wait...

The Make option

Turbo Pascal offers a solution. You can get the Make option (in the Compile menu) and Turbo Pascal to do all the work for you. The process is simple: After making any changes to any units or the main program, just Make the main program.

Turbo Pascal makes three kinds of checks.

1. *First, it checks and compares the date and time of the .TPU file for each unit used by the main program against the unit's corresponding .PAS file.* If the .PAS file has been modified since the .TPU file was created, Turbo Pascal recompiles the .PAS file, creating an updated .TPU file. So, in the first example, if you modified EDITLIB.PAS and then recompiled MYAPP.PAS (using the

Make option), Turbo Pascal would automatically recompile EDITLIB.PAS before compiling MYAPP.PAS.

2. *The second check is to see if you changed the interface portion of the modified unit.* If you did, then Turbo Pascal recompiles all other units using that unit.

   As in the second example, if you modified the interface portion of MYGLOBAL.PAS and then recompiled MYAPP.PAS, Turbo Pascal would automatically recompile MYGLOBAL.PAS, MYUTILS.PAS, EDITLIB.PAS, and GRAPHLIB.PAS (in that order) before compiling MYAPP.PAS. However, if you only modified the implementation portion, then the other dependent units don't need to be recompiled, since (as far as they're concerned) you didn't change that unit.

3. *The third check is to see if you changed any Include or .OBJ files (containing assembly language routines) used by any units.* If a given .TPU file is older than any of the Include or .OBJ files it links in, then that unit is recompiled. That way, if you modify and assemble some routines used by a unit, that unit is automatically recompiled the next time you compile a program using that unit.

The Make option has no effect on units found in TPW.TPL.

To use the Make option under the IDE, either select the Make command from the Compile menu, or press *F9*. To invoke it with the command-line compiler, use the option **/M**.

The Build option

The Build option (also in the Compile menu) is a special case of the Make option. When you compile a program using Build, it automatically recompiles *all* units used by that program (except, of course, those units in TPW.TPL). This always brings everything up to date. You can invoke Build from the command line with the **/B** option.

Conditional compilation

For a complete reference to conditional directives, refer to Chapter 21, "Compiler directives," in the Programmer's Guide.

To make your job easier, Turbo Pascal for Windows offers conditional compilation. This means that you can now decide what portions of your program to compile based on options or defined symbols.

The conditional directives are similar in format to the compiler directives you're accustomed to; in other words, they take the format

```
{$directive arg}
```

where *directive* is the directive (such as **DEFINE, IFDEF**, and so on), and *arg* is the argument, if any. Note that there *must* be a separator (blank, tab) between *directive* and *arg*. Table 5.1 lists all the conditional directives, with their meanings.

Table 5.1
Summary of compiler directives

{**$DEFINE** *symbol*}	Defines *symbol* for other directives
{**$UNDEF** *symbol*}	Removes definition of *symbol*
{**$IFDEF** *symbol*}	Compiles following code if *symbol* is defined
{**$IFNDEF** *symbol*}	Compiles following code if *symbol* is not defined
{**$IFOPT** x+}	Compiles following code if directive *x* is enabled
{**$IFOPT** x-}	Compiles following code if directive *x* is disabled
{**$ELSE**}	Compiles following code if previous **IF***xxx* is not True
{**$ENDIF**}	Marks end of **IF***xxx* or **ELSE** section

The DEFINE and UNDEF directives

The **IFDEF** and **IFNDEF** directives test to see if a given symbol is defined. These symbols are defined using the **DEFINE** directive and undefined **UNDEF** directives. (You can also define symbols on the command line and in the IDE.)

To define a symbol, insert the directive

```
{$DEFINE symbol}
```

into your program. *symbol* follows the usual rules for identifiers as far as length, characters allowed, and other specifications. For example, you might write

```
{$DEFINE debug}
```

This defines the symbol *debug* for the remainder of the module being compiled, or until the statement

```
{$UNDEF debug}
```

is encountered. As you might guess, **UNDEF** "undefines" a symbol. If the symbol isn't defined, **UNDEF** has no effect.

Defining at the command line

If you're using the command-line version of Turbo Pascal (TPCW.EXE), you can define conditional symbols on the command line itself. TPCW accepts a **/D** option, followed by a list of symbols separated by semicolons:

```
tpcw myprog /Ddebug;test;dump
```

This would define the symbols *debug*, *test*, and *dump* for the program MYPROG.PAS. Note that the **/D** option is cumulative, so that the following command line is equivalent to the previous one:

```
tpcw myprog /Ddebug /Dtest /Ddump
```

Defining in the IDE

Conditional symbols can be defined in the Conditional Defines input box (Options | Compiler). Multiple symbols can be defined by entering them in the input box, separated by semicolons. See page 153 in Chapter 6, "The IDE reference", for details about using conditional symbols in the IDE.

Predefined symbols

In addition to any symbols you define, you also can test certain symbols that Turbo Pascal for Windows has defined. Table 5.2 lists these symbols.

Table 5.2 Predefined conditional symbols

CPU86	Always defined
CPU87	Defined if an 80x87 is present at compile time
VER10	Always defined
WINDOWS	Always defined

Let's look at each in a little more detail.

The VER10 symbol

The symbol *VER10* is always defined for this version of Turbo Pascal for Windows. Future versions will have corresponding predefined symbols; for example, version 2.0 would have *VER20* defined, version 2.5 would have *VER25* defined, and so on. This allows you to create source code files that can use future enhancements while maintaining compatibility with older versions.

The WINDOWS and CPU86 symbols

These symbols are always defined (at least for Turbo Pascal for Windows running under DOS). The *WINDOWS* symbol indicates you are developing for the Windows environment. The *CPU86* symbol means you are compiling on a computer using an Intel iAPx86 (8088, 8086, 80186, 80286, 386, i486) processor.

As future versions of Turbo Pascal for other operating systems or environments and other processors become available, they will have similar symbols indicating which operating system or environment and/or processor is being used. Using these symbols, you can create a single source code file for more than one operating system or hardware configuration.

The CPU87 symbol

When you load the Turbo Pascal compiler, it checks to see if an 80x87 chip is installed. If it is, then the *CPU87* symbol is defined; otherwise, it's undefined.

The IFxxx, ELSE, and ENDIF symbols

The idea behind conditional directives is that you want to select some amount of source code to be compiled if a particular symbol is (or is not) defined or if a particular option is (or is not) enabled. The general format follows:

```
{$IFxxx}
  source code
{$ENDIF}
```

where **IF***xxx* is **IFDEF**, **IFNDEF**, or **IFOPT**, followed by the appropriate argument, and *source code* is any amount of Turbo Pascal source code. If the expression in the **IF***xxx* directive is True, then *source code* is compiled; otherwise, it is ignored as if it had been commented out of your program.

Often you have alternate chunks of source code. If the expression is True, you want one chunk compiled, and if it's False, you want the other one compiled. Turbo Pascal lets you do this with the **$ELSE** directive:

```
{$IFxxx}
  source code A
{$ELSE}
  source code B
{$ENDIF}
```

If the expression in *IFxxx* is True, *source code A* is compiled; otherwise *source code B* is compiled.

Note that all IF*xxx* directives must be completed within the same source file, which means they cannot start in one source file and end in another. However, an IF*xxx* directive can encompass an Include file:

```
{$IFxxx}
{$I file1.pas}
{$ELSE}
{$I file2.pas}
{$ENDIF}
```

That way, you can select alternate Include files based on some condition.

You can nest *IFxxx..ENDIF* constructs so that you can have something like this:

```
{$IFxxx}                                    { First IF directive }
...
{$IFxxx}                                   { Second IF directive }
...
{$ENDIF}                          { Terminates second IF directive }
...
{$ENDIF}                           { Terminates first IF directive }
```

The IFDEF and IFNDEF directives

You've learned how to define a symbol, and also that there are some predefined symbols. The **IFDEF** and **IFNDEF** directives let you conditionally compile code based on whether those symbols are defined or undefined.

It is common to use the **IFDEF** and **IFNDEF** directives to insert debugging information into your compiled code. For example, if you put the following code at the start of each unit:

```
{$IFDEF debug}
{$D+,L+}
{$ELSE}
{$D-,L-}
{$ENDIF}
```

and the following directive at the start of your program:

```
{$DEFINE debug}
```

and compile your program, then complete debugging information will be generated by the compiler for use with Turbo Debugger for Windows. In a similar fashion, you can have sections of code

that you want compiled only if you are debugging; in that case, you would write

```
{$IFDEF debug}
  source code
{$ENDIF}
```

where *source code* will be compiled only if *debug* is defined at that point.

The IFOPT directive

You may want to include or exclude code, depending upon which compiler options (range checking, I/O checking, and so on) have been selected. Turbo Pascal lets you do that with the **IFOPT** directive, which takes two forms:

```
{$IFOPT x+}
```

and

```
{$IFOPT x-}
```

See Chapter 21 in the Programmer's Guide, "Compiler directives," for a complete description.

where *x* is one of the compiler options: **$A, $B, $D, $F, $G, $I, $L, $N, $R, $S, $V, $W, $X**. With the first form, the following code is compiled if the compiler option is currently enabled; with the second, the code is compiled if the option is currently disabled. So, as an example, you could have the following:

```
var
  {$IFOPT N+}
    Radius,Circ,Area: Double;
  {$ELSE}
    Radius,Circ,Area: Real;
  {$ENDIF}
```

This selects the data type for the listed variables based on whether or not 80x87 support is enabled.

An alternate example might be

```
Assign(F,Filename);
Reset(F);
{$IFOPT I-}
IOCheck;
{$ENDIF}
```

where *IOCheck* is a user-written procedure that gets the value of *IOResult*, and prints out an error message as needed. There's no sense calling *IOCheck* if you've selected the {**$I+**} option since, if

there's an error, your program will halt before it ever calls *IOCheck*.

Optimizing code

A number of compiler options influence both the size and the speed of the code. This is because they insert error-checking and error-handling code into your program. It's best to enable them while you are developing your program, but you may want to disable them for your final version. Here are those options, with their settings for code optimization (the default settings are stated last):

- {**$A+**} enables word alignment of variables and type constants; this results in faster memory access on 80x86 systems. The default is {**$A+**}.
- {**$B-**} uses short-circuit Boolean evaluation. This produces code that can run faster, depending upon how you set up your Boolean expressions. The default is {**$B-**}.
- {**$G+**} uses additional instructions of the 80286 to improve code generation; programs compiled this way cannot run on 8088 and 8086 processors. The default is {**G-**}.
- {**$I-**} turns off I/O error-checking. By calling the predefined function *IOResult*, you can handle I/O errors yourself. The default is {**$I+**}.
- {**$N-**} generates code capable of performing all floating-point operations using the built-in 6-byte type real. When the **$N** switch is on, Turbo Pascal uses the 80x87 coprocessor or emulates the coprocessor instead using Windows software routines, depending on whether an 80x87 is present. The code generated will be able to use the four additional real types (Single, Double, Extended, and Comp). The default is {**$N-**}.
- {**$R-**} turns off range checking. This prevents code generation to check for array subscripting errors and assignment of out-of-range values. The default is {**$R-**}.
- {**$S-**} turns off stack-checking. This prevents code generation to ensure that there is enough space on the stack for each procedure or function call. The default is {**$S+**}.
- {**$V-**} turns off checking of **var** parameters that are strings. This lets you pass actual parameter strings that are of a different

length than the type defined for the formal **var** parameter. The default is {**$V+**}.

- {**$X+**} enables functions calls to be used as statements; the result of a function call can be discarded. {**$X+**} also enables the special handling of null-terminated strings. The default is {**$X+**}.

Optimizing your code using these options has two advantages. First, it usually makes your code smaller and faster. Second, it allows you to get away with something that you couldn't normally. However, they all have corresponding risks as well, so use them carefully, and reenable them if your program starts behaving strangely.

Note that besides embedding the compiler options in your source code directly, you can also set them using the Options | Compiler menu in the IDE or the */$x* option in the command-line compiler (where *x* represents a letter for a compiler directive).

C H A P T E R

6

The IDE reference

The Turbo Pascal for Windows integrated development environment (IDE) makes programming for Windows easier and more efficient. Everything you need to write, edit, compile, link, and debug your programs is at your fingertips when you start Turbo Pascal.

This chapter briefly tells you how to start and exit Turbo Pascal for Windows and then launches into detail about the individual menu items, dialog boxes, buttons, and so on. For an introduction to the basic components of the IDE, you can

- Go to Chapter 1, "Learning the new IDE." This chapter provides you with some general information about the IDE and then gets you started programming in the environment.
- Take advantage of Turbo Pascal's extensive online help system. You can get context-sensitive help in a keystroke (*F1*). If you want specific language help, move your cursor to the item you want help on and press *Ctrl+F1*, the shortcut for Help | Topic Search. The Help menu has options that display help information about all aspects of Turbo Pascal, ObjectWindows library, and the Windows Application Programming Interface (API). The help system also tells you how to use itself.

See page 162 for more about the online help system.

Starting up Turbo Pascal

Starting up Turbo Pascal is easy. From within the Windows Program Manager, double-click the Turbo Pascal icon with your mouse. If you're using your keyboard, select the Turbo Pascal icon with your cursor keys and press *Enter*.

You can also start up Turbo Pascal from the DOS command line at the same time you start up Windows. If you like, you can include the configuration file you want the IDE to use and the files you want opened. Use this syntax:

win tpw [*/Cconfig file*] *files*

For example,

```
win tpw /Cmyconfig myfile yourfile
```

will start up Windows, begin running Turbo Pascal using the settings found in the `myconfig.cfg` file, and open two windows, one containing `myfile.pas` and the other holding `yourfile.pas`.

Exiting Turbo Pascal

Shortcuts: *Alt+F4 in the CUA command set and Alt+X in the Alternate command set will exit.*

To exit Turbo Pascal, double-click the Turbo Pascal Control-menu box or press *Alt+Spacebar* to open the Control menu and select the the Close menu option. Choosing File | Exit will also exit Turbo Pascal.

Turbo Pascal Control menu

Alt Space

The Turbo Pascal Control-menu box appears on the far left of the title bar. Click it once or press *Alt+Spacebar* to display the menu. When you pull down this menu, you see several commands that affect the Turbo Pascal desktop and one that lets you switch to another application. Each edit window and dialog box also has a Control menu; see page 130 for more information about these Control menus.

Restore

The first command in the menu is Restore. Restore is available to you only if the Turbo Pascal desktop window is maximized so that it fills your entire screen or if it is minimized to an icon. When you choose this command, the window returns to its previous size.

Move

The Move command lets you move the Turbo Pascal desktop window with your keyboard. Use your arrow cursor keys to move it and press *Enter* when you are done. You will not be able to use this command if the Turbo Pascal desktop window already fills your whole screen.

Size

With your keyboard, you can change the size of the Turbo Pascal desktop window with the Size command. Use your arrow cursor keys to move the window borders. When you are satisfied with the size of your window, press *Enter*. Size is available only if the Turbo Pascal desktop is not maximized or minimized.

Minimize

Selecting the Minimize command shrinks the Turbo Pascal desktop window into the Turbo Pascal icon.

Maximize

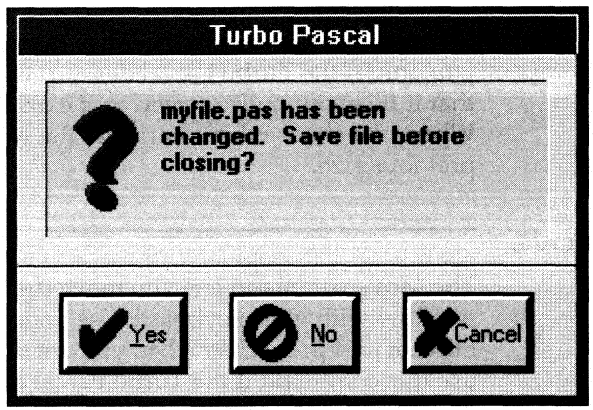
If you choose the Maximize command, the Turbo Pascal desktop window zooms to fill your entire screen. You can choose this command only if the Turbo Pascal desktop window has not been maximized already.

Close

Alt F4

The Close command closes the Turbo Pascal desktop and unloads Turbo Pascal from memory. If you have modified an edit window and not saved the text, a dialog box will appear, asking if you want to save the file:

Figure 6.1
Save file dialog box



Switch To

Ctrl Esc

Choosing the Switch To command makes the Windows Task List appear. You can use Task List to switch to another running application. Select the task you want to switch to and choose the Switch To command button.

Edit window Control menu

Alt -

Each edit window also has a Control menu when the window or its icon is active. The commands on this menu are similar to those on the Turbo Pascal Control menu.

Restore

The Restore command returns your edit window to its previous size (which is neither maximized or minimized). It will be available only if your edit window is minimized or it fills the Turbo Pascal desktop entirely.

Move

The Move command lets you move your edit window with your keyboard. After you select Move, use your arrow cursor keys to move the window to your liking and press *Enter*. Move will be available only if you edit window is not maximized.

Size

With your keyboard, you can change the size of your edit window. After selecting Size, use your arrow cursor keys to move the window borders. When you are satisfied, press *Enter*. Size will be available only if your edit window is not maximized.

Minimize

Select the Minimize command to shrink your edit window to an icon on the Turbo Pascal desktop.

Maximize

Choosing the Maximize command zooms your edit window so that it fills up the whole Turbo Pascal desktop. You will be able to select this command only if the window has not been maximized already.

Close

Ctrl F4

The Close command closes your edit window. If you have modified text in the window and have not saved the text, a dialog box will appear, asking you if you want to save your changes to the file. See Figure 6.1.

Next

Ctrl F6

This command makes the next open window or icon active.

File menu

Alt F

With the File menu, you can open and create program files in edit windows as well as save and print them.

New

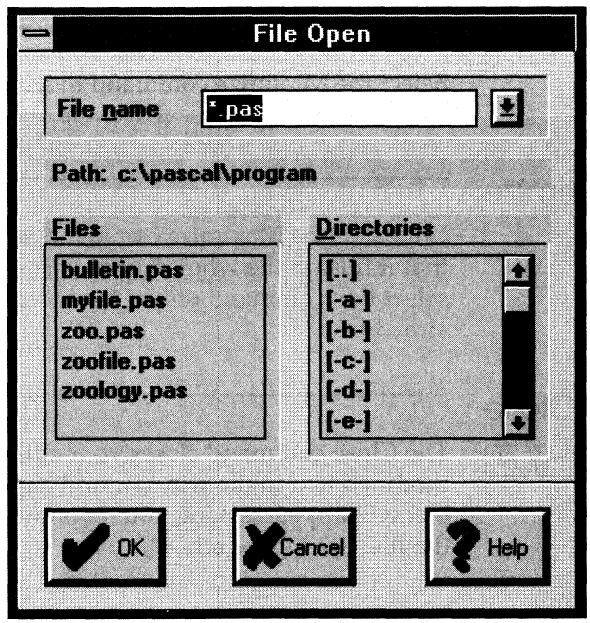
The File | New command opens a new edit window and gives it a temporary name.

Open

Alternate
F3

With the File | Open command, you can open a file you have edited and saved previously or you can create a new file with a specific name. Once you select it, the File Open dialog box appears:

Figure 6.2
File Open dialog box



The dialog box contains an input box, a file list, a directory list, command buttons labeled OK, Cancel, and Help, and a display of your current path. You can choose any of these actions:

- Type in a file name.
- Type in a file name with wildcards, which filters the file list to match your specifications when you choose OK or press *Enter*.
- Press *Alt+* ↓ or click the drop-down arrow icon next to the input box to choose a file specification from a history list of file specifications you've entered earlier. You can also just press ↓ when you are in the box to see the previous entry rather than display a history list. Continue to press ↓ to see previous file name specifications. Press ↑ to move backwards through the

list. When you are through viewing the list, press *Alt+↑* to put it away.

- View the contents of different directories by selecting a directory name in the Directories list box.

If the name of the file is listed in the Files list box on the left, double-click it and a new edit window containing that file will open. To open a file not in the current directory, double-click the directory you want in the Directories list box. If you double-click the [..] symbol, you will change to the parent directory of the subdirectory you are in.

Using your keyboard to select files and directories, press *Tab* until you reach the group box you want, use your ↑ and ↓ keys to select the item you want (pressing *Spacebar* or an arrow key will select the first item), and then press *Enter*. You can use shortcut keys (those underlined on your screen) to get to the area of the dialog box you want. For example, if you want to work with directories, press *Alt+D*, and the first directory in the Directories list box will be outlined.

If you know the name of the file you want to open or want to create a new one, you can just type in the name of the file in the File Name input box. If the file is not in your current directory, include the full path name. Finish by choosing OK.

You can also use the File Name input box to enter a file name containing standard DOS wildcard characters (* and ?). The default file name, *.PAS, displays all files with a .PAS extension in the Directory list box. If you enter another wildcard pattern, Turbo Pascal will remember it the next time you bring up the File Open dialog box during the same session.

Suppose you want to see only Turbo Pascal source files that begin with the letter `T`. You could enter this pattern,

```
T*.PAS
```

and press *Enter*. Only your Pascal source files beginning with `T` will be listed.

Save

Alternate
F2

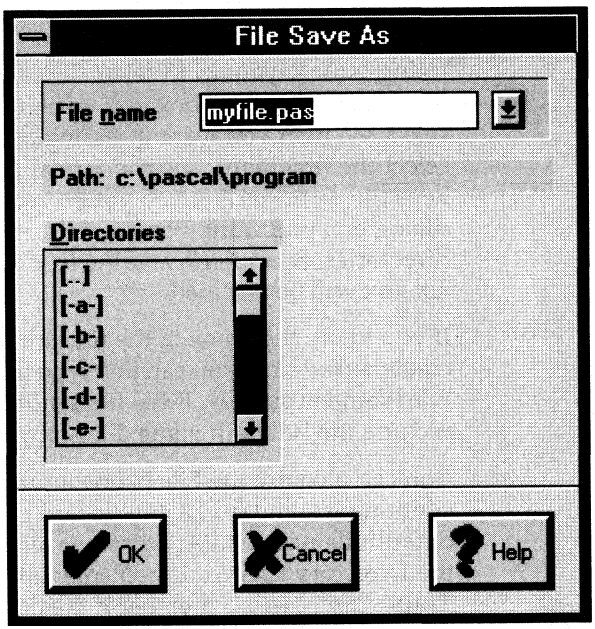
The File | Save command saves the file in the active edit window to disk. If you have not changed the default name (`NONAME00.PAS`, for example), Turbo Pascal displays the File Save As dialog box, giving you the opportunity to enter a new file name. To avoid

confusion, you should not save files with a name Turbo Pascal has given it. For example, rename `NONAME00.PAS` before saving it.

Save As

The File | Save As command saves 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 File Save As dialog box.

Figure 6.3
File Save As dialog box



Type in the new name in the File Name input box (you can include a drive and directory path), or use the Directories list to select a new path.

Save All

The File | Save All command works just like the File | Save command except that it saves the contents of all modified files, not just the file in the active edit window.

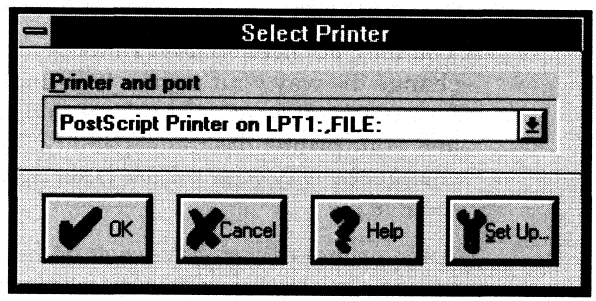
Print

The File | Print command lets you print the contents of the active edit window. Turbo Pascal expands tabs (replaces tab characters with the appropriate number of spaces) and then prints your file.

Printer Setup

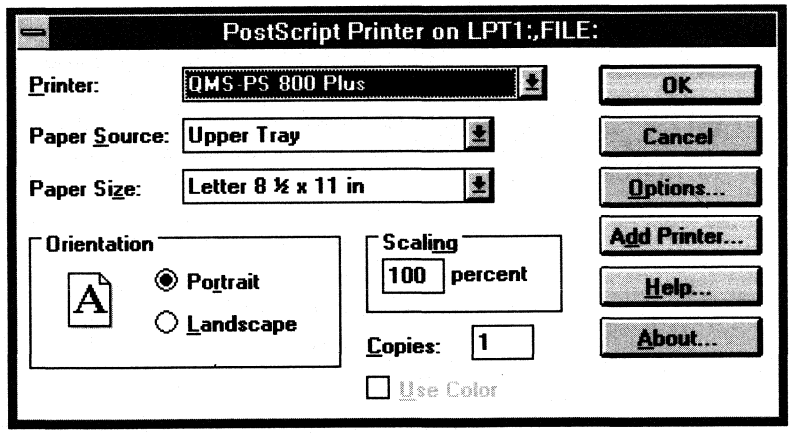
The Printer Setup command displays a dialog box you can use to set up your printer. When you installed Windows on your system, you probably also installed one or more printer drivers so you could print from Windows. The Printer Setup command lets you select which printer you want to use for printing from Turbo Pascal.

Figure 6.4
Select Printer dialog box



If you choose Setup, another dialog box appears allowing you to select a paper size, specify a particular font, and so forth. The options available to you will depend on the capabilities of your printer. This is an example of a dialog box that appears after choosing Setup:

Figure 6.5
Setup dialog box example



You may not need to use Printer Setup if you don't want to change the way your printer is normally configured. Perhaps, however, you want your printer to print on a different size paper and your printer has this capability. Turbo Pascal makes it easy to change the way your printer is set up without you having to leave the IDE.

Some printer drivers have their own help: You'll see a Help button if yours does. Choose Help for more information about setting up your printer.

If you need more help, or if there is no help for the printer-driver you are using, read the Configuring a Printer section in the "Control Panel" chapter of the *Microsoft Windows User's Guide*.

Exit

CUA

Alt F4

The File | Exit command exits Turbo Pascal and removes it from memory.

Alternate

Alt X

If you have made any changes that you haven't saved, Turbo Pascal asks you if you want to save them before exiting. To save your changes, press OK.

Closed file listing

If you have opened one or more files and then closed them, you will be able to view up to a maximum of five files at the bottom of the File menu. If you select the file name on the menu, the file will open. When you work with many open files, you can close some,

yet open them again quickly using the list, reducing clutter on your desktop.

Edit menu

Alt E

The Edit menu lets you cut, copy, paste, and clear text in edit windows.

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

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. If you want to extend a block, hold down the *Shift* key and click on the block you want to extend. Release the *Shift* key when the block is the size you want.

To select just one word, point at the word and double-click your mouse. To select an entire line, point to the line and click your mouse while pressing *Ctrl*.

To select text with your keyboard, press the *Shift* key in combination with other cursor control keys. Here is a list of key combinations that extend selected blocks in the direction indicated:

Table 6.1
Extending selected text blocks with your keyboard

Key(s)	Direction
Shift+ ←	One character to the left.
Shift+ →	One character to the right.
Shift+End	From current cursor position to the end of the current line.
Shift+Home	From current cursor position to the beginning of the current line.
Shift+ ↓	From current position to the same column on the next line.
Shift+ ↑	From current position to the same column on the previous line.
Shift+PgDn	One page down from the current position.
Shift+PgUp	One page up from the current position.
Shift+Ctrl+ ←	Left one word.
Shift+Ctrl+ →	Right one word.
Shift+Ctrl+End	To end of file.
Shift+Ctrl+Home	To beginning of file.

Once you have selected text, you are ready to cut and copy, and the *Clipboard* becomes useful.

The Clipboard is the secret behind cutting and pasting. It's a special facility available from Windows that allows information to be transferred between windows—even between the windows of different applications. The Clipboard works closely with the commands in the Edit menu.

Undo

Alt Backspace

The Edit | Undo command "undoes" your most recent edit or cursor movement. In other words, it restores the text in the active window to the way it was before your last change. Undo will insert any characters you deleted, delete any characters you inserted, replace any characters you overwrote, and move your cursor back to a prior position. If you undo a block operation, your file will appear as it was before you executed the block operation. If you continue to press Undo, Turbo Pascal will continue to undo the changes you made to the file during the current editing session.

Undo will not change an option setting that affects more than one window. For example, if you use the *Ins* key to change from Insert to Overwrite mode, then choose Undo, you won't change back into Insert mode. If you delete a character, however, switch to Overwrite mode, then select Undo, the character you just deleted

will be inserted. The effect of the operation you performed (deleting a character) is undone regardless of the mode setting.

See page 158 for more about Group Undo.

The Group Undo option in the Options | Preferences dialog box affects how Undo and its related command, Redo, will behave.

Redo

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

Cut

Shift Del

The Edit | Cut command removes the selected text from your edit window 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 until you cut or copy another block so you can paste the same text many times.

Copy

Ctrl Ins

The Edit | Copy command leaves the selected text intact in your edit window, 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 or any other Windows application that uses the Clipboard.

Paste

Shift Ins

The Edit | Paste command inserts text from the Clipboard into the current edit window at the cursor position.

Clear

Ctrl Del

The Edit | Clear command removes the selected text from your edit window but does not place the text in the Clipboard. This means you cannot paste the text as you could if you had chosen Cut or Copy. Although you can't paste the cleared text, you can undo the Clear command with Undo.

Clear is useful if you want to delete text, but you don't want to overwrite text being held in the Clipboard.

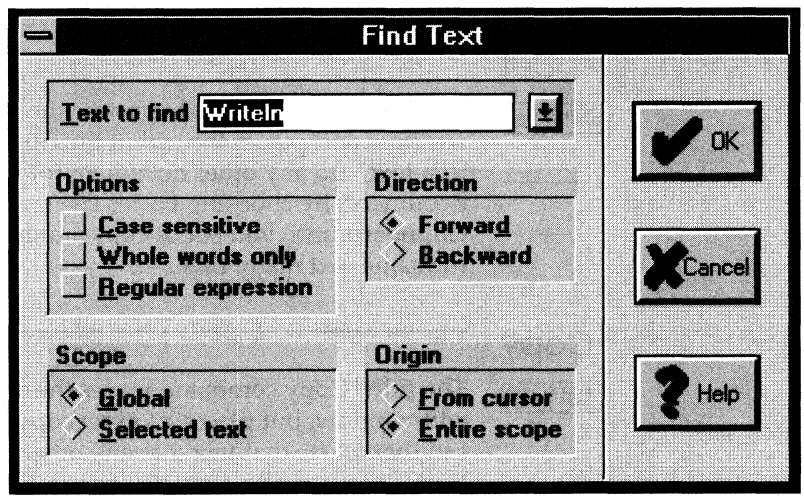
Alt S

The Search menu lets you search for text and error locations in your files.

Find

The Search | Find command displays the Find Text dialog box, in which you type in the text you want to search for and set options that affect the search.

Figure 6.6
Find Text dialog box



The Find Text dialog box contains several buttons and check boxes.

Options

You can choose from three items in the Options check boxes:

- Check the Case Sensitive box if you want Turbo Pascal to differentiate uppercase from lowercase.
- Check Whole Words Only box if you want Turbo Pascal to search for whole words only (that is, the string must have punctuation or space characters on both sides).
- Check the Regular Expression box if you want Turbo Pascal to recognize GREP-like wildcards in the search string. The wildcards are **^**, **$**, **.**, *****, **+**, **[ ]**, and \. Here's what they mean:

Table 6.2
Regular expression wildcards

^	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 character except *b*, *o*, or *t*.
[-]	A hyphen within the brackets signifies a range of characters. For example, *[b-o]* matches any character from *b* through *o*.
\	A backslash before a wildcard character tells Turbo Pascal to treat that character literally, not as a wildcard. For example, \^ matches ^ and does not look for the start of a line.

Type in the string you want to find in the input box and choose OK to begin the search, or choose Cancel to forget it. If you want to type in a string that you searched for previously, click the drop-down arrow or press *Alt+* ↓ and select the string from the displayed history list.

If the word you want to search for is displayed in your edit window, you can place your cursor on it. Turbo Pascal will pick up the word your cursor is on and display it in the Find Text dialog box.

Scope

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.

Direction

Choose from the Direction buttons to decide which direction you want Turbo Pascal to search—Forward or Backward—starting from the origin (settable with the Origin buttons).

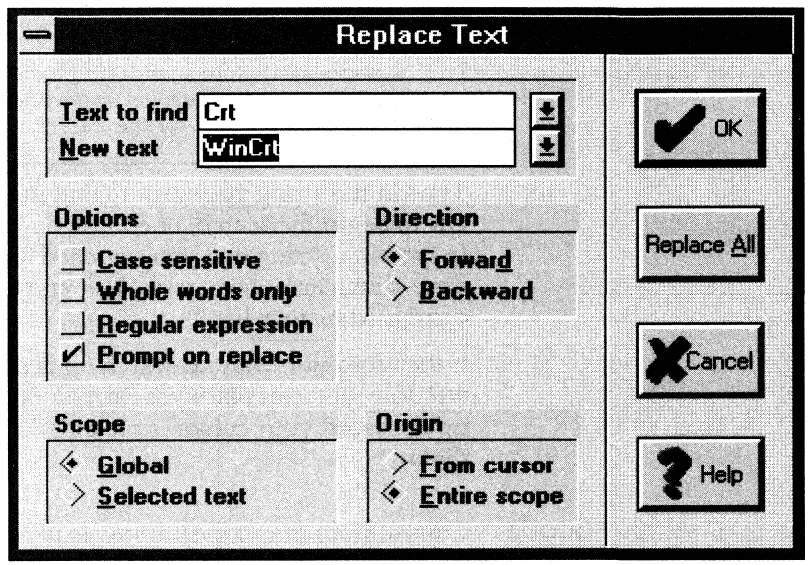
Origin

Choose from the Origin buttons to determine where the search begins. When you choose Entire Scope, the Direction buttons determine whether the search starts at the beginning or the end of the chosen scope. Choose the range of scope you want with the Scope buttons. When you chose From Cursor, the search begins from the current position of the cursor in the direction determined by the Direction buttons and for the scope set by the Scope buttons.

Replace

The Search | Replace command displays a dialog box that lets you type in the text you want to search for and the text you want to replace it with.

Figure 6.7
Replace Text dialog box



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

Type the search string and the replacement string in the input boxes and choose the OK or Replace All to begin the search, or choose Cancel to forget it. If you want to enter a string you used

previously, click the down-arrow icon or press *Alt+* ↓ to display a history list to choose from.

If Turbo Pascal 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 *Replace All*, it replaces all occurrences found, as defined by Scope, Direction, and Origin radio buttons.

Search Again

CUA

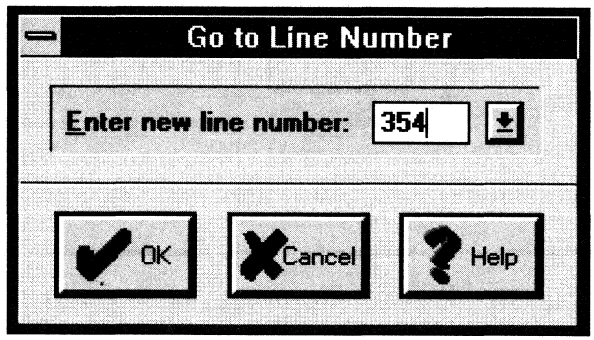
F3

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

Go to Line Number

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:

Figure 6.8
Go to Line Number dialog box



Turbo Pascal displays the current line number and column number for the active window on the status line at the bottom of the Turbo Pascal desktop window.

Show Last Compile Error

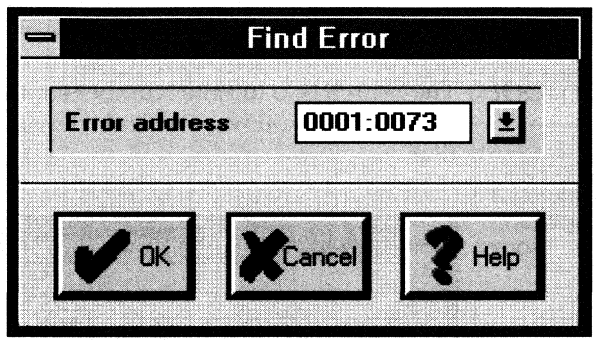
The Search | Show Last Compile Error command finds the location of the last compile error, such as a syntax error. If you select Search | Show Last Compile Error, your cursor will move to the line that caused the error. If your last error was not in the active

window, Turbo Pascal will make the window with the last compiler error active, even opening a closed file if necessary. The error number and message will appear on the status bar.

Find Error

The Search | Find Error command finds the location of your last run-time error. When the compiler tells you that your program has a run-time error, it will display the memory address of the error. Write down the address. Choose Find Error and type the address in the input box. Choose OK and the compiler will recompile your program and stop at the address location you entered, highlighting the line that caused the run-time error to occur.

Figure 6.9
Find Error dialog box



Run menu

Alt R

With the Run menu you can run your program, start up Turbo Debugger for Windows, and specify command-line parameters.

Run

Ctrl F9

The Run | Run command runs your program, using any parameters you pass to it with the Run | Parameters command. If the source code has been modified since the last compilation, Turbo Pascal will automatically remake your program.

Debugging

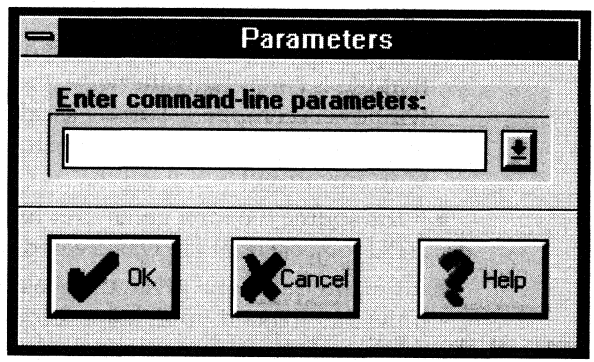
The Run | Debugger command starts Turbo Debugger for Windows so you can debug your program. Turbo Pascal tells Turbo Debugger which program to debug. If you want to use the debugger, you should check the Options | Linker | Debug Info in EXE check box before you compile and link your program. This puts the necessary debugging information in your executable file. If you haven't checked the Debug Info in EXE check box and you start Turbo Debugger, the IDE will temporarily turn on this option and recompile your program before Turbo Debugger starts so that debugging information is available.

Parameters

The Run | Parameters command lets you pass parameters to your program when you run it just as if you were running the program from the Program Manager's File | Run menu.

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

Figure 6.10
Parameters dialog box



Type in the list of parameters you want to use to run your program and choose OK. You can use any parameter that your program requires.

Compile menu

Alt C

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, and Build commands, you must have a file open in an *active* edit window or have a primary file defined (for Make and Build).

Compile

Alt F9

The Compile | Compile command compiles the file in the active edit window. While Turbo Pascal is compiling, a status box pops up to display the compilation progress and results. When compiling and linking is complete, choose OK to put the status box away. If any errors occurred, you will see the first one on the status bar and the line with the error will be highlighted.

Make

F9

The Compile | Make command creates an .EXE file, a unit, or a dynamic link library (DLL) according to the following rules.

- If a primary file has been named in the Primary File dialog box, that file is compiled; otherwise, the file in the active edit window is compiled. Turbo Pascal checks all files upon which the file being compiled depends to see if they exist and that they are current.
- If the source file for a given unit has been modified since the .TPU (object code) file was created, then that unit is recompiled.
- If the interface for a given unit has been changed, then all other units that depend upon it are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file is newer than the unit's .TPU file, then the unit is recompiled.
- If a unit contains an Include file and the Include file is newer than that unit's .TPU file, then the unit is recompiled.

If the source to a unit (.TPU file) cannot be located, that unit is *not* compiled, but is used as is.

Build

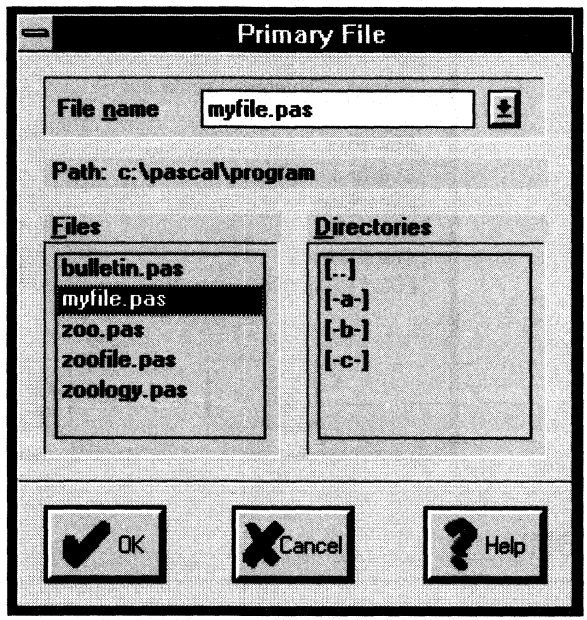
The Compile | Build command rebuilds all the components of your program, regardless of whether they're current.

This command is similar to Compile | Make except that it is unconditional. If you abort a Build 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.

Primary File

The Compile | Primary File command displays a Primary File dialog box:

Figure 6.11
Primary File dialog box



You can use the File and Directory lists to find the file you want to specify as your primary file, or type in the name of the file and choose OK.

Once you have named a primary file, you will see the file listed beside the Primary File command on the Compile menu.

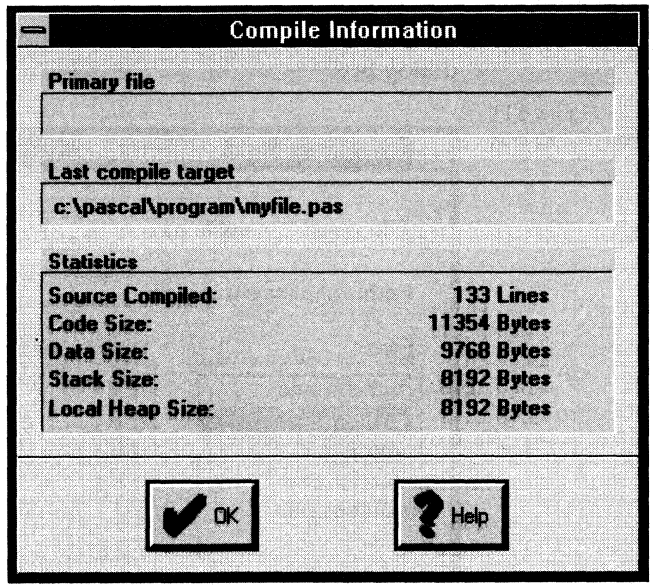
Clear Primary File

The Compile | Clear Primary File command removes the name of the primary file you specified with the Compile | Primary File command.

Information

The Compile | Information command gives you information about the last compilation.

Figure 6.12
Compile Information dialog box



Options menu

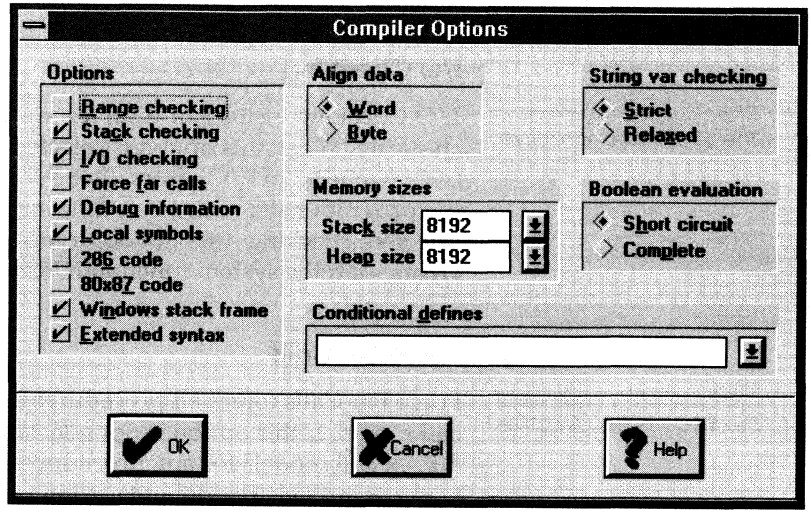
Alt O

The Options menu contains commands that let you view and change various default settings in Turbo Pascal. Most of the commands in this menu take you to dialog boxes. You can save your settings so they are in effect the next time you start up Turbo Pascal (see the Save As section on page 160 for more information).

Compiler

The Options | Compiler command displays a dialog box that displays several options that affect code compilation:

Figure 6.13
Compiler Options dialog box



The following sections describe these commands.

Options

You can use the check boxes in the Options group to tell Turbo Pascal how to compile your code. Here are descriptions of what the check box options mean:

Range Checking

*Range Checking is equivalent to the **$R** compiler directive.*

Range Checking enables or disables range checking. When this option is checked, the compiler generates code to check that array and string subscripts are within bounds, that assignments to scalar-type variables don't exceed their defined ranges, and that objects have been properly constructed. If the check fails, the program halts with a run-time error. When this box is not selected, no such checking occurs. The default setting is off.

Stack Checking

*Stack Checking is equivalent to the **$S** compiler directive.*

Stack Checking enables or disables stack checking. When this option is checked, the compiler generates code to check that space

is available for local variables on the stack before each call to a procedure or function. If the check fails, the program halts with a run-time error. When Stack checking is not selected, this checking is not done. The default setting is on.

I/O Checking

*I/O Checking is equivalent to the **$I** compiler directive.*

I/O Checking enables or disables input/output (I/O) error checking. When this option is selected, the compiler generates code to check for I/O errors after every I/O call. If the check fails, the program halts with a run-time error. When this option is not selected, checking does not occur; however, you can test for I/O errors with the system function *IOResult*. The default setting is on.

Force Far Calls

*Force Far Calls is equivalent to the **$F** compiler directive.*

Force Far Calls forces all procedures and functions to use the FAR call model. If this option is not selected, the compiler will use the NEAR call models for any procedures or functions within the file being compiled. The default setting is off.

Debug Information

*Debug Information is equivalent to the **$D** compiler directive.*

Debug Information enables or disables the generation of debugging information. This information consists of a line-number table for each procedure that maps object code addresses into source text numbers.

You can use this information with Turbo Debugger to single-step and set breakpoints to debug your program. You will usually use this option along with the Local Symbols option.

You must turn on Options | Linker | Debug Info in EXE or the compiler ignores this option setting and you will not be able to debug your program. The default setting is on.

Local Symbols

*Local Symbols is equivalent to the **$L** compiler directive.*

Local Symbols enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module; that is, the symbols in the module's implementation part, and the symbols within the module's procedures and functions. If the Local Symbols option is turned on, Turbo Debugger will use the symbol

names for debugging. (Local symbol information makes debugging much easier.) Turning off this option will free more memory for your compiles, but local symbols will not be available for use by a debugger. The default setting is on.

286 Code

*286 Code is equivalent to the **$G** compiler directive.*

80286 Code enables or disables 80286 code generation. If this option is turned off (the box is *not* checked), the compiler will generate only 8086 instructions that can run on any processor in the 80x86 family. If the option is on, Turbo Pascal uses the additional 80286 instructions to improve code generation, but the resulting program will not be able to run on 8088 or 8086 processors. This option does not check for the presence of an 80286 at run time. The default setting is off.

80x87 Code

*80x87 Code is equivalent to the **$N** compiler directive.*

The 80x87 Code check box lets you decide how you want Turbo Pascal to handle floating-point numbers. The default setting is off.

- Leave the 80x87 Code check box unchecked if you don't want Turbo Pascal to generate direct 80x87 inline code that requires an 80x87 coprocessor to run. Turbo Pascal will emulate the 80x87 if necessary with Windows software routines. You will be able to use only the 6-byte Real type in floating-point calculations.
- Choose 80x87 Code to generate direct 80x87 inline code that requires an 80x87 coprocessor to run. This option also allows you to use all Real types such as Single, Double, Extended, and Comp.

Windows Stack Frame

*The Windows Stack Frame option is equivalent to the **$W** compiler directive.*

When the Windows Stack Frame option is checked the compiler generates special prolog and epilog code for programs that run in Windows 3.0 real mode. Turn off this switch if you want your programs to run in protected mode only. The default setting is on.

Extended Syntax

*Extended Syntax is equivalent to the **$X** compiler directive.*

Extended Syntax enables or disables the use of Turbo Pascal's extended syntax, which permits user-defined function statements as well as null-terminated strings. (See the *Programmer's Guide*, Chapter 21 for more information on how this works.) If the Extended Syntax box is not checked, attempting to use these extensions will result in an error. The default setting is on.

Align Data

*Align data is equivalent to the **$A** compiler directive.*

The Align Data radio buttons let you choose how you want noncharacter data aligned. Select the Word radio button to align the data at even addresses only; choose Byte to permit the data to align at the next available address whether the address is odd or even. Word data alignment is the default.

Memory Sizes

The Memory Sizes input boxes let you configure the default memory requirements for a program.

- The Stack Size specifies the size (in bytes) of the stack segment. The default size is 8K.
- The Heap Size specifies the size (in bytes) of the local heap for local window allocations. The default size is 8K.

String Var Checking

*String var checking is equivalent to the **$V** compiler directive.*

The String Var Checking buttons allow you to choose between Strict or Relaxed string parameter error checking. If you select the Strict button, Turbo Pascal compares the formal **var**-type string parameter with the actual parameter being passed. If they are not identical, Turbo Pascal issues a compiler error. If you select the Relaxed button, this type checking does not occur and any string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Strict String Var Checking is the default.

Boolean Evaluation

*Boolean Evaluation is equivalent to the **$B** compiler directive.*

With the Boolean Evaluation buttons you choose either Short Circuit or Complete Boolean expression evaluation. Set to Short Circuit, the compiler generates code to terminate evaluation of a Boolean expression as soon as possible. For example, in the expression

```
if False and MyFunc...
```

the function *MyFunc* would never be called. If you choose Complete, the compiler will evaluate all terms in Boolean expressions.

Short circuit Boolean expression is the default setting.

Conditional Defines

Refer to Chapter 5 for more about conditional compilation.

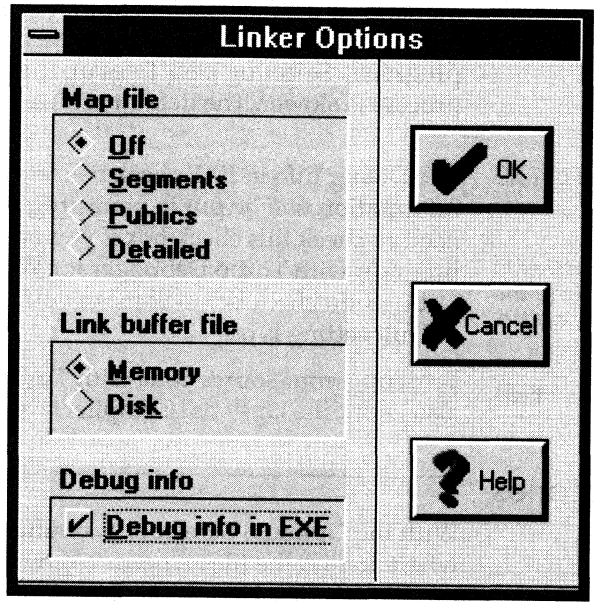
The Conditional Defines input box is used to enter define symbols that are referenced in conditional compilation directives. You can separate multiple defines with semicolons (;), for example,

```
TestCode;DebugCode
```

Linker

The Options | Linker displays a dialog box in which you can set several options that affect how your program files will be linked. The Linker dialog box looks like this:

Figure 6.14
Linker Options dialog box



This dialog box has two sets of radio buttons and a check box. The following sections contain short descriptions of what each does.

Map File

Use the Map File buttons to select the type of map file you want the linker to produce. Your choices are Off (no map file will be produced), Segments, Public, or Detailed. If you select any option other than Off, the map file is placed in the EXE and TPU directory defined in the Options | Directories dialog box.

If you choose a segment map file, the linker will produce only a list of the segments in the program, the program start address, and any warning or error messages produced during the link.

Select Publics and the linker will create the same map file as described previously, but it will also add a list of public symbols sorted alphabetically as well as in increasing address order. This type of map file is useful when you are debugging.

A Detailed map file lists segments, public symbols, a program start address, and a detailed segment map. This option gives you the most complete map file.

Link Buffer File

The Link Buffer File option tells Turbo Pascal to use either memory or disk for the link buffer. When you select Memory, linking speeds up, but you may run out of memory for large programs. Selecting Disk frees up memory, but the linking process is slower. The default setting is Memory.

Debug Info in EXE

Debug Info is equivalent to the /V command-line compiler option.

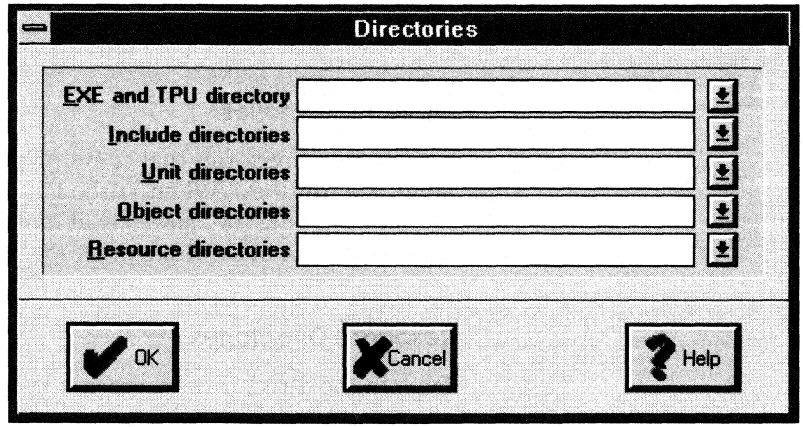
The Debug Info in EXE check box lets you select whether debug information will be put in your program's executable file. You need to check this check box if you are going to debug your program with Turbo Debugger for Windows. Unchecking the Debug Info check box will make your program smaller. The default setting is off.

To debug your source code with Turbo Debugger, make sure you also check the Options | Compiler | Debug Info in EXE check box.

Directories

With the Options | Directories command you tell Turbo Pascal where to find the files it needs to compile and link, and where to put output files. Here is the Directories dialog box:

Figure 6.15
Directories dialog box



Use the following guidelines when entering directory names in the input boxes:

- You must separate multiple directory path names (if allowed) with a semicolon (;), but don't put any whitespace between them. You can use up to a maximum of 127 characters.
- You can specify relative and absolute path names, including path names relative to the logged position in drives other than the current one. For example,

```
C:\PASCAL;C:\PASCAL\MYPROJS;A:\TPW\EXAMPLES;
```

If you enter an invalid directory name, you'll get an error message on the status bar when you try to compile a program.

The following sections describe what to put in each input box.

EXE and TPU Directory

Type in the directory where you want your executable and compiled unit files kept. Any .MAP files will be stored here if the Map File option (Options | Linker) is set to anything other than Off.

Include Directories

Specify the directories that contain your standard Include files.

Unit Directories

Type in the directories that contain your Turbo Pascal unit files.

Object Directories

Specify the directories that contain your .OBJ files (assembly language routines). When Turbo Pascal encounters the directive to link object files (**$L** *filename*), it looks first in the current directory, then in the directories specified here.

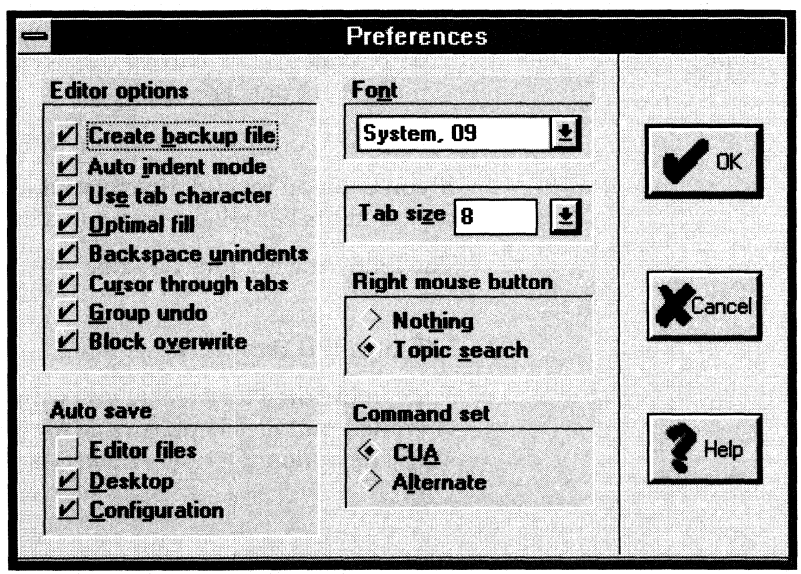
Resource Directories

Type in the directories where you keep your resource files.

Preferences

The Options | Preferences command lets you specify how you want certain aspects of the IDE to behave. Here is the Preferences dialog box:

Figure 6.16
Preferences dialog box



Editor Options

The Editor Options group has several check boxes that control how Turbo Pascal handles text in edit windows. All the default settings are on.

Create Backup File

When Create Backup File is checked (the default), Turbo Pascal automatically creates a backup of the source file in the active edit window when you choose File | Save, and gives the backup file the extension .BAK.

Auto Indent Mode

When Auto Indent Mode is checked, pressing *Enter* in an edit window positions the cursor under the first nonblank character in the preceding nonblank line. This can be a great aid in keeping your program code more readable.

Use Tab Character

When Use Tab Character is checked, Turbo Pascal inserts a true tab character when you press *Tab*. If the option is not checked, Turbo Pascal replaces tabs with spaces, the number of which is determined by the Tab Size setting, discussed earlier.

Optimal Fill

When you check Optimal Fill, Turbo Pascal begins every 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 selected.

Backspace Unindents

When Backspace Unindents is checked (the default) and the cursor is on a blank line or the first nonblank character of a line, the *Backspace* key aligns (outdents) the line to the previous indentation level.

Cursor through Tabs

When you check Cursor through Tabs, the arrow keys will move the cursor to columns within tabs; otherwise the cursor jumps several columns when cursoring over multiple tabs.

Group Undo

If you check Group Undo, when you press *Alt+Backspace* or choose Edit | Undo, the editor will undo your last group of commands. The types of commands that are grouped are insertions, deletions, overstrikes, and cursor movements. For example, if you type `OOPS` and select Undo, the entire word will be deleted. If you type the same thing and the Group Undo box is *not* checked, only the last typed character will be undone when you press Undo. You would need to press Undo four times to undo the word `OOPS`.

The Turbo Pascal editor sees inserting a carriage return (pressing *Enter*) as an insertion followed by a cursor movement. Since the type of editing changed (you inserted characters, then you moved your cursor), the grouping of inserted characters halts when you press *Enter*.

This command also affects Edit | Redo. If you select Redo after "undoing" `OOPS` in the previous example and Group undo is on, `OOPS` will appear in your window all at once. If Group Undo is off, only one letter will appear each time you choose Redo.

Block Overwrite

When Block Overwrite is checked and a block of text is marked in the active edit window, the action of typing a letter will replace the marked block with the letter typed. If Block Overwrite is not checked, typing a letter will insert the letter after the marked text.

Auto Save

If you check the Editor Files box, and if the file has been modified since the last time you saved it, Turbo Pascal automatically saves the file in the active edit window whenever you choose the Run | Run or Run | Debugger command. The default setting is off.

If you check the Desktop box, Turbo Pascal saves the way you have your Turbo Pascal desktop set up when you exit. When you start another session, your desktop will appear as you left it. The Desktop auto save option also saves your desktop in its current state when you switch to a new configuration file; it then reloads your desktop settings when you switch back to the old configuration file. The default setting is on.

If you check the Configuration box, Turbo Pascal saves the name of your primary file and all the settings you made with the Options menu when you exit. When you start another session, all

the options you selected in the previous session will be in effect. The default setting is on.

Font

For more information about fonts, see your Microsoft Windows User's Guide.

You can change the size and appearance of the text in your edit windows by selecting a new font. Open the list box to see your choices. Windows supplies a selection of screen fonts. When you installed your printer, any screen fonts your printer can use were automatically installed. If you purchase additional fonts from a font or printer vendor, you may get additional screen fonts. Turbo Pascal displays and can use only fixed-pitch screen fonts such as Courier. Select the font you want and then when you choose OK to leave the dialog box, the new font will take effect.

Tab Size

The Tab Size input box controls how many columns the cursor moves for each tab stop. Legal values are 2 through 16; the default is 8.

To change the way tabs are displayed in a file, change the tab size value to the size you prefer. Turbo Pascal redisplays all tabs in that edit window in the size you chose.

Right Mouse Button

Turbo Pascal uses the right button to perform a topic search. When you press your right mouse button, a help window will open displaying language help on the item your cursor was on in the active edit window. You can change this by selecting Nothing. The default is Topic Search.

Command Set

Chapter 7 details the differences between the two command sets.

The Command Set lets you determine if your editor will use Common User Access (CUA) commands as other Windows editors do, or an Alternate command set similiar to that used by editors in other Borland language products.

Open

If you want to use settings other than those found in the current configuration file, you can select the Options | Open command to retrieve the new settings. Type in the name of the configuration file containing the settings you want and choose OK.

This option can also create a new configuration with the name you specify.

See page 113 for information about clearing your desktop while using the same configuration file.

If Auto Save is on, opening an existing configuration file saves your current settings in the old configuration file. If you open a new configuration file, the old desktop settings disappear but the current configuration settings become part of the new configuration file until you change them.

Save

In your current configuration file, the Options | Save command saves the name of the primary file and all the settings you selected with the Options menu as well as the current state of your desktop. To save your options to a different configuration file, use the Options | Save As command.

The name of your current configuration file appears next to the Options | Save command.

Save As

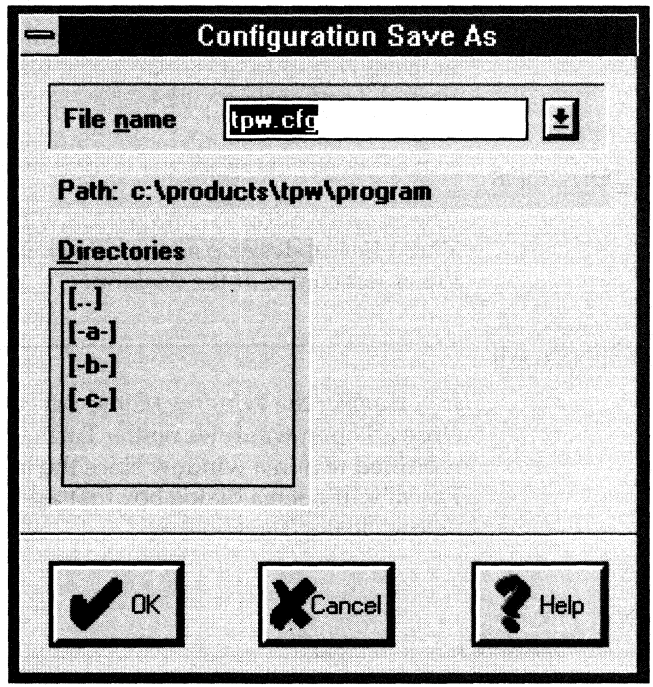
The Options | Save As command brings up the dialog box shown in Figure 6.17.

With the Configuration Save As dialog box you can choose to save the file name of the primary file and settings you selected with the Options menu in a configuration file. The default configuration file name is `tpw.cfg`.

You can have more than one configuration file, each with different settings. To create a new configuration file, make the settings you want, choose the Options | Save As command to display the Configuration Save As dialog box, type in a new file name and choose OK. If you omit the file extension, Turbo Pascal will automatically add `.cfg`. For example, `my` becomes `my.cfg`.

The next time you start up Turbo Pascal, all your settings will be in effect. If you have chosen to auto-save the Configuration file in the Preferences dialog box, all settings you make during a session will automatically be saved to the current configuration file when you exit Turbo Pascal for Windows.

Figure 6.17
Configuration Save As



Window menu

Alt W

The Window menu lists some window management commands. (For information about other window management commands, see page 12.) You open windows with the File | Open or File | New commands.

At the bottom of the Window menu, you will see a list of open windows. If there is more than one, you can switch to another window and make it active by selecting it with either your mouse or your keyboard. This is especially handy if other windows on the desktop cover up the window you want to switch to.

Tile

Shift F5

Choose Window | Tile to tile your open windows (arrange them so they do not overlap one another, but together they cover the entire desktop).

Cascade

Shift F4

Choose Window | Cascade to stack all open windows (overlapping windows of the same size with only part of each underlying window visible).

Arrange Icons

Choosing Window | Arrange Icon will rearrange any icons on the Turbo Pascal desktop so they are evenly spaced, beginning at the lower left corner of the desktop.

Close All

If you select the Window | Close All command, Turbo Pascal will close all open windows on the Turbo Pascal desktop. If you have modified text in a window since the last time you saved it, Turbo Pascal will open a dialog box that asks you if want to save the text before closing the window.

Help menu

Alt H

The Help menu gives you access to online help in a help window. There is help information on virtually all aspects of the IDE and Turbo Pascal. You use the help system in Turbo Pascal just as you would use help in Windows itself or any other Windows application. Most of the items on this menu describe the type of help available: Compiler Directives, ObjectWindows, Procedures and Functions, Reserved Words, Standard Units, Turbo Pascal Language, and Windows API.

The following sections provide details on the other help items: Index, Topic Search, Glossary, Using Help, and About Turbo Pascal.

Index

Shift F1

The Help | Index command displays an index to all the help entries in the help system. You can scroll the list. When you find a keyword that interests you, choose it by cursoring to it and pressing *Enter* or double-clicking it.

Topic Search

Ctrl F1

The 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 wiindow and choose Topic Search. You can get help on things like procedure names, reserved words, and so on. If an item is not in the help system, the help index displays the closest match. You can set up the right button of your mouse to do a topic search with the Options | Preferences dialog box.

Glossary

The Help | Glossary command displays definitions about the IDE and Turbo Pascal terms in alphabetical order. When you find a term you want defined, click it with your mouse or press *Tab* until it is highlighted and then press *Enter*.

Using Help

The Help | Using Help command displays information on how to use Turbo Pascal's help system.

About Turbo Pascal

Choosing the Help | About Turbo Pascal command will display a dialog box that identifies the version of Turbo Pascal and displays the Borland copyright notice.

C H A P T E R

7

The editor from A to Z

You probably know how to edit text with the Turbo Pascal for Windows editor already. When you start up Turbo Pascal for the first time, you'll find the editor behaves like other Windows editors. Since the editor follows Common User Access (CUA) guidelines, the same editing commands you use in other Windows applications will work in Turbo Pascal as well. You have little new to learn.

Remember, this chapter is concerned *just* with the editor. For a tutorial about the editor and the IDE, refer to Chapter 1; for an in-depth discussion of the whole Turbo Pascal integrated environment, refer to Chapter 6.

Command sets

If you are a long-time Borland language user, you might find yourself pining for the editing commands you used in the DOS world. We have a solution for you.

The Turbo Pascal for Windows editor has two command sets: the Common User Access (CUA) command set that makes the editor behave like other Windows editors, and the Alternate command set that turns the editor into a Borland-style editor. If you have used Borland language products before, you may feel more comfortable with the Alternate command set.

When you start up Turbo Pascal for the first time, the editor will use the CUA command set. To use the Alternate command set, choose Options | Preferences and select the Alternate radio button from the Command Set group. Choose OK.

Changing to a different command set not only changes the commands you use to perform various editing tasks, but it also changes a few menu shortcuts. Here are the differences:

Table 7.1
Menu shortcuts that change

Menu command	CUA	Alternate
File | Open		*F3*
File | Save		*F2*
File | Exit	*Alt+F4*	*Alt+X*
Search | Search Again	*F3*	

The Edit menu

The Edit menu contains commands for cutting, copying, and pasting in a file and for undoing and redoing edit operations. When you first start Turbo Pascal, an edit window is already active. To open other edit windows, go to the File menu and choose Open. From an edit window, press *Alt+E* to go directly to the Edit menu; 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.

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; you can move back and forth in your text with the scroll bars in your edit window.

Editor reference

The Turbo Pascal editor is sophisticated and powerful. In addition to the menu choices, it has many commands to move the cursor around, page through text, find and replace strings, and so on.

Most of these commands need no explanation. Those that do are described in the text following Table 7.2.

Table 7.2
Editing commands

*A word is defined as a sequence of characters separated by one of the following: space < > , ; . ( ) { } ^ ` * + - / $ # = | ~ ? ! " % & ': @ \, and all control and graphic characters.*

Command	Both modes	CUA	Alternate
Cursor movement commands			
Character left	←		Ctrl+S
Character right	→		Ctrl+D
Word left	Ctrl+ ←		Ctrl+A
Word right	Ctrl+ →		Ctrl+F
Line up	↑		Ctrl+E
Line down	↓		Ctrl+X
Scroll up one line	Ctrl+W		
Scroll down one line	Ctrl+Z		
Page up	PgUp		Ctrl+R
Page down	PgDn		Ctrl+C
Beginning of line	Home		Ctrl+Q S
End of line	End		Ctrl+Q D
Top of window		Ctrl+E	Ctrl+Q E
Bottom of window		Ctrl+X	Ctrl+Q X
Top of file	Ctrl+Home		Ctrl+Q R Ctrl+PgUp
Bottom of file	Ctrl+End		Ctrl+Q C Ctrl+PgDn
Insert and delete commands			
Delete character	Del		Ctrl+G
Delete character to left	Backspace		Ctrl+H
Delete line	Ctrl+Y		
Delete to end of line		Shift+Ctrl+Y	Ctrl+Q Y
Delete word	Ctrl+T		
Insert line	Ctrl+N		
Insert mode on/off	Ins		Ctrl+V
Block commands			
Copy to Clipboard	Ctrl+Ins		
Cut to Clipboard	Shift+Del		
Delete block	Ctrl+Del		
Indent block		Shift+Ctrl+I	Ctrl+K I
Paste from Clipboard	Shift+Ins		
Read block from disk		Shift+Ctrl+R	Ctrl+K R
Unindent block		Shift+Ctrl+U	Ctrl+K U
Write block to disk		Shift+Ctrl+W	Ctrl+K W
Extending selected blocks			
Left one character	Shift+ ←		
Right one character	Shift+ →		
End of line	Shift+End		
Beginning of line	Shift+Home		
Same column on next line	Shift+ ↓		
Same column on previous line	Shift+ ↑		
One page down	Shift+PgDn		
One page up	Shift+PgUp		
Left one word	Shift+Ctrl+ ←		

Table 7.2: Editing commands (continued)

Command	Both modes	CUA	Alternate
Right one word	*Shift+Ctrl+ →*		
End of file	*Shift+Ctrl+End*		*Shift+Ctrl+PgD*
Beginning of file	*Shift+Ctrl+Home*		*Shift+Ctrl+PgU*
Other editing commands			
Autoindent mode on/off			*Ctrl+O I*
Current compiler options		*Ctrl+O*	*Ctrl+O O*
Cursor through tabs on/off			*Ctrl+O R*
Exit Turbo Pascal		*Alt+F4*	*Alt+X*
Find place marker		*Ctrl n* *	*Ctrl+Q n* *
Help	*F1*		
Help index	*Shift+F1*		
Insert control character	*Ctrl+P***		
Maximize window			*F5*
Open file			*F3*
Optimal fill mode on/off			*Ctrl+O F*
Pair matching		*Alt+[, Alt+]*	*Ctrl+Q [,* *Ctrl+Q]*
Save file			*F2* *Ctrl+K S*
Search			*Ctrl+Q F*
Search again		*F3*	
Search and replace			*Ctrl+Q A*
Show last compile error			*Ctrl+Q W*
Set marker		*Shift+Ctrl n* *	*Ctrl+K n* *
Tabs mode on/off			*Ctrl+O T*
Topic search help	*Ctrl+F1*		
Undo	*Alt+Backspace*		
Unindent mode on/off			*Ctrl+O U*

* *n* represents a number from 0 to 9.

** Enter control characters by first pressing *Ctrl+P*, then pressing the desired control character.

Block commands

A block of text is any amount of text, from a single character to hundreds of lines, that is selected on your screen. There can be only one block in a window at a time. Select a block just as though you were editing with any other Windows editor—with your mouse or by holding down *Shift* while moving your cursor to the end of the block with the arrow keys. Once selected, the block can be copied, moved, deleted, or written to a file. You can use the Edit menu commands to perform these operations or you can use the keyboard commands listed in the following table.

When you choose Edit | Copy or press *Ctrl+Ins*, the selected block is copied to the Clipboard. When you choose Edit | Paste or *Shift+Ins*, the block held in the Clipboard is pasted at the current cursor position. The selected text remains unchanged and is no longer selected.

If you choose Edit | Cut or press *Shift+Del*, the selected block is moved from its original position and held in the Clipboard. It is pasted at the current cursor position when you choose the Paste command.

The copying, cutting, and pasting commands are the same in both the CUA and Alternate command sets.

Table 7.3: Block commands in depth

Command	CUA	Alternate	Function
Copy block	*Ctrl+Ins, Shift+Ins*	*Ctrl+Ins, Shift+Ins*	Copies a previously selected block to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The original block is unchanged. If no block is selected, nothing happens.
Copy text	*Ctrl+Ins*	*Ctrl+Ins*	Copies selected text to the Clipboard.
Cut text	*Shift+Del*	*Shift+Del*	Cuts selected text to the Clipboard.
Delete block	*Ctrl+Del*	*Ctrl+Del*	Deletes a selected block. You can "undelete" a block with Undo.
Move block	*Shift+Del, Shift+Ins*	*Shift+Del, Shift+Ins*	Moves a previously selected block from its original position to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The block disappears from its original position. If no block is marked, nothing happens.
Paste from Clipboard	*Shift+Ins*	*Shift+Ins*	Pastes the contents of the Clipboard.
Read block from disk	*Shift+Ctrl+R*	*Ctrl+K R*	Reads a disk file into the current text at the cursor position exactly as if it were a block. The text read is then selected 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.
Write block to disk	*Shift+Ctrl+W*	*Ctrl+K W*	Writes a selected block to a file. 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 .PAS). If you prefer to use a file name without an extension, append a period to the end of its name.

If you have used Borland editors in the past, you may prefer to use these block commands while using the Alternate command set:

Table 7.4
Borland-style block commands

Command	Keys	Function
Turn on text selection	*Ctrl+K B*	Begins the selection of text. Text selection ends with copying (*Ctrl+K K*) or cutting (*Ctrl+K V*) to the Clipboard, or turning text selection off with *Ctrl+K H*.
Turn off text selection	*Ctrl+K H*	Stops the selection of text and the selected text becomes unselected.
Copy text to Clipboard	*Ctrl+K K*	Copies the selected text to the Clipboard.
Cut text to Clipboard	*Ctrl+K V*	Cuts the selected text to the Clipboard.
Paste from Clipboard	*Ctrl+K C*	Pastes the contents of the Clipboard into your active edit window.

Changing your mind: Undo

The editor has an Undo command that takes the pain out of making mistakes or changing your mind. To reverse your previous editing operation, choose Edit | Undo or simply press *Alt+Backspace*. If you continue to choose Undo, the editor continues to reverse actions until your file returns to the state it was in when you began your current editing session. You can even "undo" Undo with the Edit | Redo command. Undo works in both CUA and Alternate modes. For more information about using Undo and Redo, see Chapter 6, "The IDE reference."

Other editing commands

The next table describes certain editing commands in more detail. The table is arranged alphabetically by command name.

Table 7.5: Other editor commands in depth

Command	CUA	Alternate	Function
Auto indent		*Ctrl+O I*	Toggles the automatic indenting of successive lines. You can also use Options I Preferences I Auto Indent in the IDE to turn automatic indenting on and off.
Compiler options	*Ctrl+O*	*Ctrl+O O*	Inserts the current compiler option settings at the head of your active edit window.
Cursor thru tabs		*Ctrl+O R*	The arrow keys will move the cursor to the middle of tabs when this option is on; otherwise the cursor jumps several columns when cursoring over multiple tabs. *Ctrl+O R* is a toggle.
Find place marker	*Ctrl+n**	*Ctrl+Q n**	Finds up to ten place markers (*n* can be any number in the range 0 to 9) in text. Move the cursor to any previously set marker by pressing *Ctrl+Q* and the marker number.
Open file		*F3*	Lets you load an existing file into an edit window.
Optimal fill		*Ctrl+O F*	Toggles optimal fill. Optimal fill begins every line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters.
Save file		*F2*	Saves the file and returns to the editor.
Set place	*Shift+Ctrl n**	*Ctrl+K n**	Mark up to ten places in text. After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the Find Place Marker command (being sure to use the same marker number). You can have ten places marked in each window.
Show last compile error		*Ctrl+Q W*	Highlights the last syntax error the compiler found during the last compile. The error message appears on the status bar. If you have already closed that file, Turbo Pascal will reopen it and highlight the error.
Tab mode		*Ctrl+O T*	Toggles Tab mode. You can specify the use of true tab characters in the IDE with the Options I Preferences I Use Tab Character option.
Unindent		*Ctrl+O U*	Toggles Unindent. You can turn Unindent on and off from the IDE with the Options I Preferences I Backspace Unindents option.

* *n* represents a number from 0 to 9.

Searching and searching again

To search for a string of text in the active edit window, follow these instructions:

1. Choose Search | Find or press *Ctrl+Q F*. This opens the Find Text dialog box.
2. Type the string you are looking for into the Text to Find input box.
3. You can also set various search options:
   - The Options check boxes determine whether the search will be case sensitive for whole words only, and for regular expressions.
   - The Scope radio buttons control how much of the file you search.
   - The Direction radio buttons control whether you do a forward or backward search.
   - The Origin radio buttons control where the search begins.
4. Use *Tab*, shortcut keys, or your mouse to cycle through the options. Use ↑ and ↓ to set the radio buttons and *Space* to toggle the check boxes.
5. Finally, choose the OK button to carry out the search or the Cancel button to cancel. Turbo Pascal performs the operation.
6. If you want to search for the same item repeatedly, use Search | Search Again or press *Ctrl+L*.

Search and replace

1. Choose Search | Replace. This opens the Replace Text dialog box.
2. Type the string you are looking for into the Text to find input box.
3. Press *Tab*, shortcut keys, or use your mouse to move to the New Text input box. Type in the replacement string.
4. You can set the same search options as in the Find Text dialog box.
5. Finally, choose OK or Replace All to begin the search, or choose Cancel to cancel. Turbo Pascal performs the operation. Choosing Replace All will replace every occurrence found.
6. If you want to stop the operation, choose Cancel at any point when the search has paused.

Pair matching

There you are, debugging your source file that is full of functions, parenthesized expressions, and a whole slew of other constructs that use delimiter pairs. In fact, your file is riddled with

- braces: { and }
- parentheses: (and)
- brackets: [and]
- double quotes: "
- single quotes: '
- parentheses and asterisks: (* and *)

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. With Turbo Pascal'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.
2. To locate the mate to this selected delimiter, simply press *Alt+[* if you are using the CUA command set, or *Ctrl+Q [* if you are using the Alternate command set.
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 the intervening code contains no unmatched delimiters of that type. If it moves to the wrong delimiter, you know you have made a mistake; now all you need to do is track down the source of the problem.

We've told you the basics of Turbo Pascal's pair matching commands; now you need some details. The following section covers these points:

- There are actually two match-pair editing commands: one for forward matching (*Alt+[*) and the other for backward matching (*Alt+]*). (The commands will be *Ctrl+Q [* and *Ctrl+Q]* if you are using the Alternate command set.)
- If there is no mate for the delimiter you've selected, the editor doesn't move the cursor.

Directional and nondirectional matching

Opening braces, brackets, closing braces, and parentheses are all directional; the editor knows which way to search for the mate, so it doesn't matter which match pair command you give.

Double and single quotes are not directional. You must specify the correct match pair command.

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 parenthesis (( ). 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 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 direction by giving the correct match pair command. If you give the command *Alt+[*, the editor searches forward for the match; if you give the command *Alt+]*, it searches backward for the match.

The following table summarizes the delimiter pairs, whether they imply search direction, and whether they are nestable:

Table 7.6
Delimiter pairs

Nestable delimiters are explained after this table.

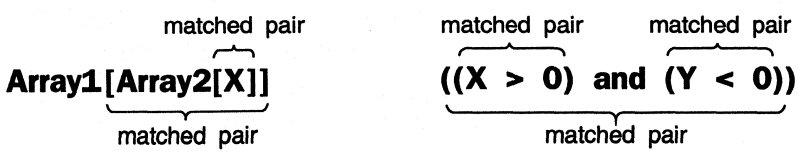
Delimiter pair	Direction implied?	Are they nestable?
{ }	Yes	No
( )	Yes	Yes
[ ]	Yes	Yes
(* *)	Yes	No
" "	No	No
' '	No	No
(. .)	Yes	Yes

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 an example:

Figure 7.1
Search for match to square bracket or parenthesis



C H A P T E R

8

The command-line compiler

Turbo Pascal for Windows command-line compiler (TPCW.EXE) lets you invoke all the functions of the IDE compiler (TPW.EXE) from the DOS command line.

You run TPCW.EXE from the DOS prompt using a command line with the following syntax:

TPCW [*options*] *file* [*options*]

options are zero or more optional parameters that provide additional information to the compiler. *file* is the name of the source file to compile. If you type `TPCW` alone, it displays a help screen of command-line options and syntax.

If *file* does not have an extension, TPCW assumes .PAS. If you don't want the file you're compiling to have an extension, you must append a period (.) to the end of *file*. If the source text contained in *file* is a program, TPCW creates an executable file named FILENAME.EXE. If *file* contains a unit, TPCW creates a Turbo Pascal unit file named FILENAME.TPU; if *file* contains a dynamic-link library, TPCW creates a library file named FILENAME.DLL.

You can specify a number of options for TPCW. An option consists of a slash (*/*) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such as a number, a symbol, or a directory name. Options can be given in any order and can come before and/or after the file name.

Compiler options

The IDE allows you to set various options through the menus; TPCW gives you access to most of these same options using the slash (/) command. You can also precede options with a hyphen (-) instead of a slash (/), but those options that start with a hyphen must be separated by blanks. For example, the following two command lines are equivalent and legal:

```
TPCW -IC:\TPW -DDEBUG SORTNAME -$S- -$F+
TPCW /IC:\TPW/DDEBUG SORTNAME /$S-/$F+
```

The first uses hyphens with at least one blank separating options; the second uses slashes and no separation is needed.

Table 8.1 lists all the command-line options that have integrated environment equivalents. A few do not have corresponding IDE menu commands.

Table 8.1
Command-line options

Command Line	Menu Command	Setting
/$A+	O | C | Align Data	Word
/$A-	O | C | Align Data	Byte
/$B+	O | C | Complete Boolean Eval	Complete
/$B-	O | C | Complete Boolean Eval	Short Circuit
/$D+	O | C | Debug Information	On
/$D-	O | C | Debug Information	Off
/$F+	O | C | Force Far Calls	On
/$F-	O | C | Force Far Calls	Off
/$G+	O | C | 286 Code	On
/$G-	O | C | 286 Code	Off
/$I+	O | C | I/O Checking	On
/$I-	O | C | I/O Checking	Off
/$L+	O | C | Local Symbols	On
/$L-	O | C | Local Symbols	Off
/$M*stack,heap*	O | C | Memory Sizes	
/$N+	O | C | 80x87 Code	On
/$N-	O | C | 80x87 Code	Off
/$R+	O | C | Range Checking	On
/$R-	O | C | Range Checking	Off
/$S+	O | C | Stack Checking	On
/$S-	O | C | Stack Checking	Off
/$V+	O | C | String Var Checking	Strict
/$V-	O | C | String Var Checking	Relaxed
/$W+	O | C | Windows Stack Frame	On
/$W-	O | C | Windows Stack Frame	Off
/$X+	O | C | Extended Syntax	On
/$X-	O | C | Extended Syntax	Off
/B	Compile | Build	
/D*defines*	O | C | Conditional Defines	
/E*path*	O | D | EXE & TPU Directory	
/F*segment:offset*	S | Find Error	
/GS	O | L | Map File	Segments

Table 8.1: Command-line options (continued)

/GP	O|L|Map File	Public
/GD	O|L|Map File	Detailed
/I*path*	O|D|Include Directories	
/L	O|L|Link Buffer	Disk
/M	Compile|Make	
/O*path*	O|D|Object Directories	
/Q	(none)	
/R*path*	O|D|Resource Directories	
/T*path*	(none)	
/U*path*	O|D|Unit Directories	
/V	O|L|Debug Info	On

Compiler directive options

Turbo Pascal supports several compiler directives, all of which are described in Chapter 21 of the *Programmer's Guide*, "Compiler directives." When embedded in the source code, these directives take one of the following forms:

```
{$directive+}
{$directive-}
{$directive info}
```

The **/$** and **/D** command-line options allow you to change the default states of most compiler directives. Using **/$** and **/D** on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The switch directive option

The **/$** option allows you to change the default state of the following switch directives: **$A, $B, $D, $F, $G, $I, $L, $N, $R, $S, $V,** and **$X**. The syntax of a switch directive option is **/$** followed by the directive letter, followed by a plus (**+**) or a minus (**-**). For example,

```
TPCW MYSTUFF /$R-
```

would compile MYSTUFF.PAS with range checking turned off, while

```
TPCW MYSTUFF /$R+
```

would compile it with range checking turned on. Note that if a {**$R+**} or {**$R-**} compiler directive appears in the source text, it overrides the **/$R** command-line option.

You can repeat the **/$** option in order to specify multiple compiler directives:

```
TPCW MYSTUFF /$R-/$I-/$V-/$F+
```

Alternately, TPCW allows you to write a list of directives (except for **$M**), separated by commas:

Note that only one dollar sign ($) is needed.

```
TPCW MYSTUFF /$R-,I-,V-,F+
```

In addition to changing switch directives, **/$** also allows you to specify a program's memory allocation parameters, using the following format:

```
/$MSTACK,HEAP
```

where *stack* is the stack size, and *heap* is the local heap size. Both values are in bytes, and each is a decimal number unless it is preceded by a dollar sign (**$**), in which case it is assumed to be hexadecimal. So, for example, the following command lines are equivalent:

```
TPCW MYSTUFF /$M16384,4096
TPCW MYSTUFF /$M$4000,$1000
```

Note that, because of its format, you cannot use the **$M** option in a list of directives separated by commas.

The conditional defines option

The **/D** option lets you define conditional symbols, corresponding to the {**$DEFINE** *symbol*} compiler directive or the O|C| Conditional Defines option in the IDE. The **/D** option must be followed by one or more conditional symbols, separated by semicolons (;). For example, the following command line

```
TPCW MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, *iocheck*, *debug*, and *list*, for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple **/D** directives, you can concatenate the symbol lists. Thus

```
TPCW MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

is equivalent to the first example.

Compiler mode options

A few options affect how the compiler itself functions. These are **/M** (Make), **/B** (Build), **/L** (Link Buffer) and **/Q** (Quiet). As with the other options, you can use the hyphen format (remember to separate the options with at least one blank).

The make (/M) option

TPCW has a built-in MAKE utility to aid in project maintenance. The **/M** option instructs TPCW to check all units upon which the file being compiled depends.

A unit will be recompiled if

- the source file for that unit has been modified since the .TPU file was created.
- any file included with the **$I** directive, or any .OBJ file linked in by the **$L** directive, is newer than the unit's .TPU file.
- the interface section of a unit referenced in a **uses** statement has changed.

Units in TPW.TPL are excluded from this process.

If you were applying this option to the previous example, the command would be

```
TPCW MYSTUFF /M
```

The build all (/B) option

You can't use /M and /B at the same time.

Instead of relying on the **/M** option to determine what needs to be updated, you can tell TPCW to update *all* units upon which your program depends using the **/B** option. This is the same as Compile | Build.

If you were using this option in the previous example, the command would be

```
TPCW MYSTUFF /B
```

The find error option

When a program terminates due to a run-time error, it displays an error code and the address (*segment:offset*) at which the error occurred. By specifying that address in a **/F***segment:offset* option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the **$D** compiler directive).

Suppose you have a file called TEST.PAS that contains the following program:

```
program Test;
var
  x : Real;
begin
  x := 0;
  x := x div x;                    { Force a divide by zero error }
end.
```

First, compile this program using the command-line compiler:

```
TPCW TEST
```

If you do a DIR TEST.*, DOS lists two files: TEST.PAS, your source code, and TEST.EXE, the executable file.

Now, type `WIN TEST` to start Windows and run TEST. You'll get a run-time error: "Run-time error 200 at 0000:0018." Notice that you're given an error code (200) and the address (0000:0018 in hex) of the instruction pointer (CS:IP) where the error occurred. To figure out which line in your source caused the error, simply invoke the compiler, use **/F** and specify the segment and offset as reported in the error message:

```
C:\>TPCW TEST /F0:18
Turbo Pascal for Windows Copyright (c) 1983,91 Borland International
TEST.PAS(7)
TEST.PAS(6): Target address found.
  x := x div x;
  ^
```

In order for TPCW to find the run-time error with **/F**, you must compile the program with all the same command-line parameters you used the first time you compiled it.

The compiler now gives you the file name and line number, and points to the offending line number and text in your source code.

As mentioned previously, you *must* compile your program and units with debug information enabled for TPCW to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a {**$D-**} compiler directive or a **/$D-** option, TPCW will not be able to locate run-time errors.

The link buffer option

The **/L** option disables buffering in memory when .TPU files are linked to create an .EXE file. Turbo Pascal's built-in linker makes two passes. In the first pass through the .TPU files, the linker marks every procedure that gets called by other procedures. In the second pass, it generates an .EXE file by extracting the marked procedures from the .TPU files.

This is the same as the Disk setting (O I L I Link Buffer File)

By default, the .TPU files are kept in memory between the two passes; however, if the **/L** option is specified, they are read again from disk during the second pass. The default method is faster but requires more memory; for very large programs, you may have to specify **/L** to link successfully.

The quiet option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When TPCW is invoked with the quiet mode option

```
TPCW MYSTUFF /Q
```

its output is limited to the sign-on message and the usual statistics at the end of compilation. If an error occurs, it will be reported.

Directory options

TPCW supports several options that allow you to specify the six directory lists used by TPCW: Turbo, EXE & TPU, Include, Unit, Object, and Resource.

The Turbo Directory option

TPCW looks for two files when it is executed: TPCW.CFG, the configuration file, and TPW.TPL, the resident library file. TPCW automatically searches the current directory and the directory containing TPCW.EXE. The **/T** option lets you specify other directories in which to search. For example, you could say

```
TPCW /TC:\TPW\BIN MYSTUFF
```

If you want the **/T** option to affect the search for TPCW.CFG, it must be the very first command-line argument, as in the previous example.

The EXE & TPU directory option

This option lets you tell TPCW where to put the .EXE and .TPU files it creates. It takes a directory path as its argument:

```
TPCW MYSTUFF /EC:\TPW\BIN
```

This is the same as the O I D I EXE & TPU Directory command.

If no such option is given, TPCW creates the .EXE and .TPU files in the same directories as their corresponding source files.

The include directories option

Turbo Pascal supports Include files through the {**$I** *filename*} compiler directive. The **/I** option lets you specify a list of directories in which to search for Include files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPCW to search for Include files in C:\TPW\INCLUDE and D:\INC *after* searching the current directory:

This is the same as O I D I Include Directories command.

```
TPCW MYSTUFF /IC:\TPW\INCLUDE;D:\INC
```

If multiple **/I** directives are specified, the directory lists are concatenated. Thus

```
TPCW MYSTUFF /IC:\TPW\INCLUDE/ID:\INC
```

is equivalent to the first example.

The unit directories option

This is the same as the O I D I Unit Directories command.

When you compile a program that uses units, TPCW first attempts to find the units in TPW.TPL (which is loaded along with TPCW.EXE). If they cannot be found there, TPCW searches for *unitname*.TPU in the current directory. The **/U** option lets you specify additional directories in which to search for units. As with the previous options, you can specify multiple directory paths as long as you separate them with semicolons (;). For example, the following command line causes TPCW to look in C:\TP\UNITS and C:\LIBRARY for any units it doesn't find in TPW.TPL or the current directory:

```
TPCW MYSTUFF /UC:\TPW\UNITS;C:\LIBRARY
```

As with the **/I** option, if multiple **/U** options are specified, the directory lists are concatenated.

The object files directories option

This is the same as the O I D I Object Directories command.

Using {**$L** *filename*} compiler directives, Turbo Pascal allows you to link in .OBJ files containing external assembly language routines, as explained in Chapter 22, "The inline assembler," in the *Programmer's Guide*. The **/O** option lets you specify a list of directories in which to search for such .OBJ files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPCW to search for .OBJ files in C:\TPW\ASM and D:\OBJECT *after* searching the current directory:

```
TPCW MYSTUFF /OC:\TP\ASM;D:\OBJECT
```

Like the **/I** option, if multiple **/O** options are specified, the directory lists are concatenated.

The resource directories option

This is the same as the O I D I Resource Directories command.

With the **/R** option, you tell Turbo Pascal where to look for .RES files you created to be linked into your program. For example, the following command causes TPCW to search from resource files in C:/TPW\RESOURCE and D:\RES *after* searching the current directory:

```
TPCW MYSTUFF /RC:\TPCW\RESOURCE;D:\RES
```

As with the **/I** option, if multiple **/R** options are specified, the directory lists are concatenated.

Debug options

TPCW has two command-line options that enable you to generate debugging information for Borland's Turbo Debugger for Windows.

The map file option

Unlike the binary format of .EXE and .TPU files, a .MAP file is a legible text file that can be output on a printer or loaded into the editor.

The **/G** option, like the O | L | Map File option, instructs TPCW to generate a .MAP file that shows the layout of the .EXE file. The **/G** option must be followed by the letter *S*, *P*, or *L* to indicate the desired level of information in the .MAP file. A .MAP file is divided into three sections:

- **S**egment
- **P**ublics
- **L**ine Numbers

The **/GS** option outputs only the Segment section, **/GP** outputs the Segment and Publics section, and **/GD** outputs all three sections.

For modules (program and units) compiled in the {**$D+,L+**} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the {**$D+,L-**} state, only symbols defined in a unit's **interface** part are listed in the Publics section.

For modules compiled in the {**$D-**} state, there are no entries in the Line Numbers section.

The debugging option

When you specify the **/V** option on the command line, TPCW appends Turbo Debugger-compatible debug information at the end of the .EXE file. Turbo Debugger for Windows includes both source- and machine-level debugging, powerful breakpoints (including breakpoints with conditionals or expressions attached to them), and it lets you debug huge applications via virtual

machine debugging on a 386 or two-machine debugging (connected via the serial port).

This is the same as checking the O I L I Debug Info in EXE box.

Even though the debug information generated by **/V** makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and if it is executed, the .EXE file does not require additional memory.

The extent of debug information appended to the .EXE file depends on the setting of the **$D** and **$L** compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the {**$D+,L+**} state, which is the default, *all* constant, variable, type, procedure, and function symbols become known to the debugger. In the {**$D+,L-**} state, only symbols defined in a unit's **interface** section become known to the debugger. In the {**$D-**} state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

The TPCW.CFG file

You can set up a list of options in a configuration file called TPCW.CFG, which will then be used in addition to the options entered on the command line. Each line in TPCW.CFG corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a TPCW.CFG file, you can change the default setting of any command-line option.

TPCW allows you to enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a TPCW.CFG file, you can still override them on the command line.

When TPCW starts, it looks for TPCW.CFG in the current directory. If the file isn't found there, and if you are running DOS 3.*x* or higher, TPCW looks in the directory where TPCW.EXE resides. To force TPCW to look in a specific list of directories (in addition to the current directory), specify a **/T** command-line option as the first option on the command line.

If TPCW.CFG contains a line that does not start with a slash (**/**) or a hyphen (**-**), that line defines a default file name to compile. In that case, starting TPCW with an empty command line (or with a command line consisting of command-line options only and no

file name) will cause it to compile the default file name, instead of displaying a syntax summary.

Here's an example TPCW.CFG file, defining some default directories for include, object, and unit files, and changing the default states of the **$F** and **$S** compiler directives:

```
/IC:\TPW\INC;C:\TPW\SRC
/OC:\TPW\ASM
/UC:\TPW\UNIT
/$F+
/$S-
```

Now, if you type

```
TPCW MYSTUFF
```

at the system prompt, TPCW acts as if you had typed in the following:

```
TPCW /IC:\TPW\INC;C:\TPW\SRC /OC:\TPW\ASM /UC:\TPW\UNIT /$F+ /$S-
MYSTUFF
```

I N D E X

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Notes