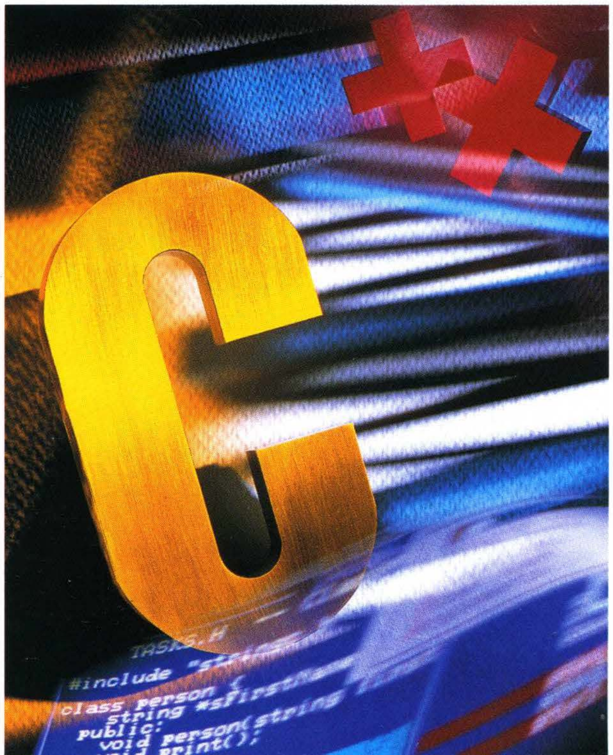
TURBO C®++ GETTING STARTED

■ INSTALLATION ■ TUTORIALS ■

BORLAND

# TURBO C®++

GETTING STARTED



BORLAND

# Turbo C®++

# Getting Started

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001

This manual was produced with Sprint®: The Professional Word Processor

PRINTED IN THE USA.
R1 10 9 8 7 6 5 4 3 2

# C O N T E N T S

# T A B L E S

# F I G U R E S

# I N T R O D U C T I O N

Turbo C++ is for C++ and C programmers who want a fast, efficient compiler; for Turbo Pascal programmers who want to learn C++ or C with all the "Turbo" advantages; and for anyone just learning C++ or C. Turbo C++ is also for anyone who wants both AT&T's C++ version 2.0 and ANSI C.

*Turbo C++ is highly compatible with existing Turbo C code.*

C++ is an object-oriented programming (OOP) language. It's the next step in the natural evolution of C. It is portable, so you can easily transfer application programs written in C++ from one system to another. You can use C++ for almost any programming task, anywhere.

## What's in Turbo C++

Turbo C++ includes many of the latest features users ask for:

*Chapter 1 tells you how to install Turbo C++. Chapter 2 tells you to where you can find out more about each of these features.*

- C++: Turbo C++ offers you the full power of C++ programming (implementing C++ version 2.0 from AT&T). To help you get started, we're also including C++ class libraries. To help you make the transition from C++ version 1.2, we've included support for version 1.2 streams.
- ANSI C: Turbo C++ provides you with an up-to-date implementation of the latest ANSI C standard.
- Borland's new Programmer's Platform. The Programmer's Platform is a new generation user interface; it goes beyond the old integrated environment to provide access to the full range of programs and tools on your computer. It includes:
  - mouse support
  - multiple overlapping windows
  - a multi-file editor

- support for inline assembler code

and much more.

- VROOMM (Virtual Run-time Object-Oriented Memory Manager): VROOMM lets you overlay your code without complexity. You select the code segments for overlaying; VROOMM takes care of the rest, doing the work needed to fit your code into 640K.
- An online tour of the new programmer's platform.
- Online hypertext help, with copy-and-paste program examples for practically every function.
- Many indispensable library functions, including heap checking functions and a complete set of complex and BCD math functions.

Other features include:

- An enhancement to the **–S** option: Now your C source code is added as comments to the resultant assembler code.
- Far objects and huge arrays.
- Several new pragmas and warnings: an ill-formed pragma warning, an **argsused** pragma, and a start-up pragma.
- Alternate .CFG files. You can create several and use the one that suits your needs at any given time.
- Response files for the command-line compiler.

## Hardware and software requirements

Turbo C++ runs on the IBM PC family of computers, including the XT, AT, and PS/2, along with all true IBM compatibles. Turbo C++ requires DOS 2.0 or higher and at least 640K; it runs on any 80-column monitor. The minimum requirement is a hard disk drive and one floppy drive.

Turbo C++ includes floating-point routines that let your programs make use of an 80x87 math coprocessor chip. It emulates the chip if it is not available. Though it is not required to run Turbo C++, the 80x87 chip can significantly enhance your programs' performance.

Turbo C++ also supports a mouse. Though the mouse isn't required, if you have one, you must have one of the following for full compatibility:

- Microsoft Mouse version 6.1 or later, or any mouse compatible with this mouse
- Logitech Mouse version 3.4 or later
- Mouse Systems' PC Mouse version 6.22 or later
- IMSI mouse version 6.11 or later

## The Turbo C++ implementation

Turbo C++ is a full implementation of the AT&T C++ version 2.0. It is also American National Standards Institute (ANSI) C standard and fully supports the Kernighan and Ritchie definition. In addition, Turbo C++ includes certain extensions for mixed-language and mixed-model programming that let you exploit your PC's capabilities. See Chapter 1, "The Turbo C++ language standard," in the *Programmer's Guide* for a complete description of Turbo C++.

## The Turbo C++ package

Your Turbo C++ package consists of a set of distribution disks and four manuals:

*Getting Started and the User's Guide tell you how to use this product; the Programmer's Guide and the Library Reference focus on programming in C.*

- *Turbo C++ Getting Started* (this manual)
- *Turbo C++ User's Guide*
- *Turbo C++ Programmer's Guide*
- *Turbo C++ Library Reference*

In addition to the four manuals, you'll find a convenient *Quick Reference* booklet. The distribution disks contain all the programs, files, and libraries you need to create, compile, link, and run your Turbo C++ programs; they also contain sample programs, several standalone utilities, a context-sensitive help file, an integrated debugger, and additional C documentation not covered in these guides.

### *Getting Started*

This volume introduces you to Turbo C++ and shows you how to create and run both C and C++ programs. It consists of information you'll need to get up and running quickly: installation, tutorials, primers, and a guide to the Turbo C++ documentation set. These are the chapters in this manual:

**Chapter 1: Installing Turbo C++** tells you how to install Turbo C++ on your system.

**Chapter 2: Navigating the Turbo C++ manuals** introduces some of Turbo C++ most interesting features; where appropriate, it tells you where to find out more about them.

*You might also want to try the online tutorial, TCTOUR.*

**Chapter 3: Learning the new IDE** walks you through the integrated environment and introduces the new editor, mouse support, and other new or changed features. This is a light overview of the Turbo C++ system. In-depth information can be found in Chapter 1, "The IDE reference," in the *User's Guide.*

**Chapter 4: An introduction to C** is an overview of the C language. This chapter introduces you to the elements of C programs, data and data types, operators, functions, arrays, structures, and other aspects of the C language.

*Chapters 5 and 6 work together: one is theory, the other is practice.*

**Chapter 5: A C++ primer** is an introduction to the concepts of object-oriented programming using C++.

**Chapter 6: Hands-on C++** is a swift hands-on introduction to C++.

**Chapter 7: Debugging in the new IDE** introduces you to the Turbo C++ integrated debugger and walks you through sample programs with built-in bugs to demonstrate various features of the debugger.

The **Bibliography** contains a listing of books relating to generic C and C++, and to Turbo C++ specifically.

## The *User's Guide*

The *User's Guide* provides reference chapters on the features of Turbo C++: Borland's new integrated environment, including the greatly enhanced editor and Project Manager, as well as details on using Turbo C++'s utilities, command-line compiler, and customization program.

**Chapter 1: The IDE reference** provides a complete reference to the integrated development environment.

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

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

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

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

**Chapter 6: Customizing Turbo C++** tells how to use the TCINST program to customize Turbo C++. You can adjust onscreen colors, editor defaults, compiler and linker defaults, and many other aspects of Turbo C++ with this program.

**Appendix A: Turbo Editor macros** describes the Turbo Editor Macro Language, a powerful utility you can use to enhance or change the Turbo C++ editor.

## The *Programmer's Guide*

The *Programmer's Guide* provides useful material for the experienced C user: a complete language reference for C and C++, a cross-reference to the run-time library, C++ streams, memory models, mixed-model programming, video functions, floating-point issues, and overlays, plus error messages.

**Chapter 1: The Turbo C++ language standard** describes the Turbo C++ language. Any extensions to the ANSI C standard are noted here. This chapter is basically a language reference and syntax for both the C and C++ aspects of Turbo C++.

**Chapter 2: Run-time library cross-reference** provides some information on the source code for the run-time library, lists and describes the header files, and provides a cross-reference to the run-time library, organized by subject. For example, if you want to find out which functions relate to graphics, you would look in this chapter under the topic "Graphics."

**Chapter 3: C++ streams** tells you how to use the C++ version 2.0 stream library. The earlier version stream library is documented online.

**Chapter 4: Memory models, floating point, and overlays** covers memory models, mixed-model programming, floating-point concerns, and overlays.

**Chapter 5: Video functions** is devoted to handling text and graphics in Turbo C++.

**Chapter 6: Interfacing with assembly language** tells how to write assembly language programs so they work well when called from Turbo C++ programs.

**Chapter 7: Error messages** lists and explains all run-time and compiler-generated errors and warnings, and suggests possible solutions.

**Appendix A: ANSI implementation-specific standards** describes those aspects of the ANSI C standard that have been left loosely defined or undefined by ANSI. These aspects will vary, then, according to each implementation. This appendix tells how Turbo C++ operates in respect to each of these aspects.

### The *Library Reference*

The *Library Reference* contains a detailed list and explanation of Turbo C++'s extensive library functions and global variables.

**Chapter 1: The run-time library** is an alphabetically arranged reference to all Turbo C++ library functions. Each entry gives syntax, include files, an operative description, return values, and portability information for the function, and a reference list of related functions.

**Chapter 2: Global variables** defines and discusses Turbo C++'s global variables. You can use these to save yourself a great deal of programming time on commonly needed variables (such as dates, time, error messages, stack size, and so on).

## 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 `TC` to start up Turbo C++). |
| ALL CAPS | We use all capital letters for the names of constants and files. |
| [ ] | 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.* |
| < > | Angle brackets in the function reference section enclose the names of include files. |
| **Boldface** | Turbo C++ function names (such as **printf**) and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used in text, but not in program examples, for Turbo C++ reserved words (such as **char**, **switch**, **near**, and **cdecl**), for format specifiers and escape sequences (**%d**, **\t**), and for command-line options (**/A**). |
| *Italics* | Italics indicate variable names (identifiers) that appear in text. They can represent terms that you can use as is, or that you can think up new names for (your choice, usually). They are also used to emphasize certain words, such as new terms. |
| *Keycaps* | This typeface indicates a key on your keyboard. It is often used to describe a particular key you should press. (For example, "Press *Esc* to exit a menu.") |
| ⌨ | This icon indicates keyboard actions. |
| 🖱 | This icon indicates mouse actions. |

## 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 B (Turbo Prolog & Turbo C)" 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 C++

1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001, USA

*See the README file included with your distribution disks for details on how to report a bug.*

You can also telephone our Technical Support department between 6 am and 5 pm 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. The version number for Turbo C++ is displayed when you first load the program and before you press any keys.
3. Computer brand, model, and the brands and model numbers of any additional hardware.
4. Operating system and version number. (The version number can be determined by typing `VER` at the MS-DOS prompt.)
5. Contents of your AUTOEXEC.BAT file.
6. Contents of your CONFIG.SYS file.

C H A P T E R

1

# Installing Turbo C++

Your Turbo C++ package includes two different versions of the Turbo C++ compiler: the integrated environment version and a command-line version. You *must* use the INSTALL program to install Turbo C++ on your system; it automatically copies files from the distribution disks to your hard disk. There is no copy protection. For reference, the README file on the installation disk includes a list of the distribution files.

*If you don't already know how to use DOS commands, refer to your DOS reference manual before setting up Turbo C++ on your system.*

We assume you are already familiar with DOS commands. For example, you'll need the DISKCOPY command to make backup copies of your distribution disks. Make a complete working copy of the distribution disks when you receive them, then store the original disks away in a safe place.

If you are not familiar with Borland's No-Nonsense License Statement, read the agreement included with your Turbo C++ package. Be sure to mail us your filled-in product registration card; this guarantees that you'll be among the first to hear about the hottest new upgrades and versions of Turbo C++.

This chapter contains the following information:

- installing Turbo C++ on your system
- accessing the README file
- accessing the HELPME! file
- a pointer to more information on Turbo Calc

Once you have installed Turbo C++, you'll be ready to start digging into Turbo C++. But certain chapters and manuals were written with particular programming needs in mind. Chapter 2, "Navigating the Turbo C++ manuals," tells where to find out more about Turbo C++'s features in the manuals.

## Installing Turbo C++

Turbo C++ has an automatic installation program called INSTALL. You *must* use INSTALL. This program detects what hardware you are using and configures Turbo C++ appropriately. It also creates directories as needed and transfers files from your distribution disks (the disks you bought) to your hard disk. Its actions are self-explanatory; the following text tells you all you need to know.

To install Turbo C++:

1. Insert the installation disk into drive A. Type the following command, then press *Enter.*
2. `A:INSTALL`
3. Press *Enter* at the installation screen.
4. Follow the prompts.

***Important!***

When it is finished, the INSTALL program reminds you to read the latest about Turbo C++ in the README file, which contains important, last-minute information about Turbo C++. The HELPME!.DOC file also answers many common technical support questions.

Also, once you have installed Turbo C++, you'll get a chance to try out TCTOUR. TCTOUR is a guided tour of some of the highlights of the new Turbo C++ integrated environment. TCTOUR is in the TOUR subdirectory (off of your Turbo C++ directory).

*To exit Turbo C++, press Alt-X.*

Once you have installed Turbo C++ and tried out TCTOUR, and if you're anxious to get up and running, change to the Turbo C++ directory and type `TC`. Press *Enter.* Otherwise, continue reading this chapter and the next for important start-up information.

After you have tried out the Turbo C++ integrated environment, you may want to permanently customize some of the options. The TCINST program makes this easy to do. See Chapter 6, "Customizing Turbo C++," in the *User's Guide* for instructions.

## Laptop systems

If you have a laptop computer (one with an LCD or plasma display), in addition to carrying out the procedures given in the previous sections, you need to set your screen parameters before using Turbo C++. The Turbo C++ integrated development environment version (TC.EXE) works best if you type `MODE BW80` at the DOS command line before running Turbo C++.

Although you could create a batch file to take care of this for you, you can also easily install Turbo C++ for a black-and-white screen with the Turbo C++ customization program, TCINST. See Chapter 6, "Customizing Turbo C++," in the *User's Guide* for instructions. With this customization program, choose "Black and White" from the **S**creen Modes menu.

## The README file

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

To access the README file:

1. If you haven't installed Turbo C++, insert your Turbo C++ disk into drive A. If you have installed Turbo C++, skip to step 3 or go on to the next paragraph.
2. Type `A:` and press *Enter*.
3. Type `README` and press *Enter*. Once you are in README, use the ↑ and ↓ keys to scroll through the file.
4. Press *Esc* to exit.

*See Chapter 3, "Learning the new IDE," in this volume, and Chapter 1, "The IDE reference" in the User's Guide, for more details on using the new integrated environment.*

If you've already installed Turbo C++, you can open README into an edit window, following these steps:

1. Start Turbo C++ by typing `TC` on the command line. Press *Enter*.
2. Press *F10*. Choose **File | Open**. Type in `README` and press *Enter*. Turbo C++ opens the README file in an edit window.
3. When you're done with the README file, choose **File | Quit** (or continue playing with the new environment).

## The HELPME!.DOC file

Your installation disk contains a file called HELPME!.DOC, which contains answers to problems that users commonly run into. Consult it if you find yourself having difficulties. You can use the README program to look at HELPME!.DOC. Type this at the command line:

```
README HELPME!.DOC
```

## Turbo Calc

Your Turbo C++ package includes the source code for a spreadsheet program called Turbo Calc. Before you compile it, read the online documentation (TCALC.DOC) for it.

C H A P T E R

2

# Navigating the Turbo C++ manuals

This chapter accomplishes two things:

- It tells you briefly about Turbo C++'s hottest features: what they are, the concepts behind them, how to use them.
- It tells you where in these manuals you can find out more about the new features and other aspects of Turbo C++.

If you read the instructions on how to install Turbo C++ on page 10, you also learned how to start Turbo C++ and how to exit from it. If not, and if you want to just jump right in and start programming, refer back to that page.

## Features

Turbo C++ has many powerful features, listed on page 1. This section tells you a little more about some of these features, and points you to where you can go for in-depth information on them.

### C++

With Turbo C++, you get two compilers in one. You get all the capabilities of ANSI C, *plus* all the capabilities of C++. Chapters 5 and 6, "A C++ primer" and "Hands-on C++," provide the theory of C++ programming and a tutorial.

In addition to this, we've included a ready-made set of C++ class libraries for you to use. These libraries use classes to perform a variety of functions for you. So some advantages of C++, such as extensibility and reusability, are yours immediately.

### VROOMM (overlays)

*Chapter 4, "Memory models, floating point, and overlays," in the Programmer's Guide covers overlays in depth.*

Turbo C++'s VROOMM (Virtual Run-time Object-Oriented Memory Manager) gives you intelligent overlays, unlike any overlay scheme you may have used before. If you are already familiar with overlays in another (non-Borland) product, you have some pleasant surprises coming. First, VROOMM can determine how and when to overlay, thus relieving you of that task. Second, since VROOMM is based on a set of highly sophisticated algorithms, it is much faster and more efficient than other overlay schemes.

### Borland's new integrated environment

Because Borland's integrated environment has been completely redone, we recommend that you take the guided tour provided by TCTOUR even if you are already familiar with other Borland products. First change directories to the TOUR subdirectory (in your TC directory). Then type `TCTOUR` and press *Enter*.

For more, read Chapter 3, "Learning the new IDE." This chapter gives hands-on experience with a range of features in the new integrated environment, including mouse support, the help system, the clipboard, new ways to handle windows, editing multiple files, transferring out to other programs (and then back into Turbo C++), and so on. Chapter 1, "The IDE reference," in the *User's Guide* is a reference to every aspect of the new integrated environment.

## Using the manuals

The manuals are arranged so that you can pick and choose among the books and chapters to find exactly what you need to know at the time you need to know it. *Getting Started* and the *User's Guide* provide information on how to use Turbo C++ as a product; the *Programmer's Guide* and the *Library Reference* provide material on programming issues in C and C++.

Five chapters in this volume provide introductions to and tutorials on subjects of interest:

- Chapter 3, "Learning the new IDE"
- Chapter 4, "An introduction to C"

- Chapter 5, "A C++ primer"
- Chapter 6, "Hands-on C++"
- Chapter 7, "Debugging in the new IDE"

As mentioned earlier, the integrated environment is brand new, so you might want to browse through the first chapter on this list even if you are familiar with Turbo C or Turbo Pascal.

The chapters of the *User's Guide* are for use as reference chapters to using Turbo C++:

*Refer to these chapters as needed after you have worked through the appropriate introductions and tutorials.*

- Chapter 1, "The IDE reference"
- Chapter 2, "Managing multi-file projects"
- Chapter 3, "The editor from A to Z"
- Chapter 4, "The command-line compiler"
- Chapter 5, "Utilities"
- Chapter 6, "Customizing Turbo C++"
- Appendix A, "Turbo Editor macros"

## New programmers or programmers learning C

*The bibliography lists useful sources for further information on C.*

If you are learning C, start with chapters 3 and 4, "Learning the new IDE" and "An introduction to C," in this book. These chapters introduce you to the Turbo C++ integrated environment and give you an overview of programming in C. Chapters 5 and 6, "A C++ primer" and "Hands-on C++," give a brief introduction to programming in C++. Depending on how much you want to be on the leading edge of technology and on how confident you feel about programming in C, you may want to jump right into those chapters after reading Chapter 4. Chapter 7, "Debugging in the new IDE," runs you through the Turbo C++ integrated debugger. Later, you can use chapters 1 through 6 in the *User's Guide* for reference.

Your next step is to start programming in C. For the most part, you will be using the *Library Reference*. Or, you might prefer to use the online help; it contains much of the same information as the *Library Reference*, and includes programming examples for almost every function that you can copy into your own program.

## Experienced C programmers

If you are an experienced C programmer and you've already installed Turbo C++, you'll probably want to jump immediately to the *Programmer's Guide* and to the *Library Reference*. If, however, you want to learn more about the new Turbo C++ integrated environment, including the integrated debugger, read chapters 3 and 7 ("Learning the new IDE" and "Debugging in the new IDE") in this book.

If you are interested in C++, read chapters 5 and 6, "A C++ primer"and "Hands-on C++.

The *Programmer's Guide* covers certain useful programming issues, such as C++ streams, assembly language interface, memory models, video functions, overlays, and far and huge pointers. In addition, the *Programmer's Guide* provides a cross-reference to the *Library Reference* by functionality (Chapter 2, "Run-time library cross-reference"). So, for example, if you want to know which C functions are associated with graphics, you would turn to that chapter and look up the subject "Graphics."

C H A P T E R

3

# Learning the new IDE

***Note:** Before you run this tutorial, you must install Turbo C++ with the default subdirectories. See Chapter 1 for details.*

In this chapter, you'll get hands-on experience with Turbo C++'s new version of the Borland integrated development environment (IDE). Even if you're already familiar with other Borland products, we recommend that you work through this chapter. Turbo C++ introduces the new IDE; this chapter gives you an overview of some of the most important new features. This tutorial consists of three lessons:

- **Lesson 1** shows how to start Turbo C++ and how to load, edit, and save files.
- **Lesson 2** covers how to compile and run a program from within the IDE.
- **Lesson 3** exits you from Turbo C++.

If your fingers are itchy and you want to get up and running on Turbo C++ *right now,* go to Chapter 1 on page 10. Otherwise, read through this tutorial to learn the basics of using the IDE. It should take you about 30 minutes to finish.

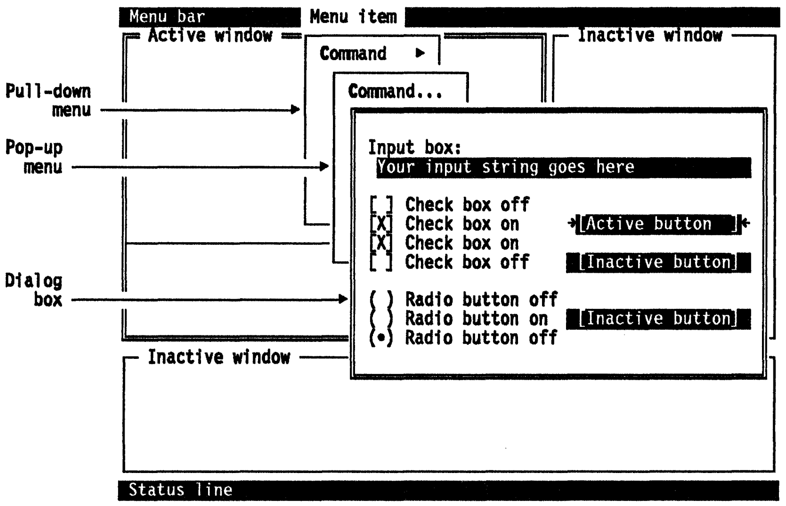
If you want in-depth information about specific items on the menus or in the dialog boxes, refer to Chapter 1, "The IDE reference," in the *User's Guide.* If you'd like to step through an online, interactive tour of the IDE, run TCTOUR.EXE (type `tctour` at the DOS prompt while in the TOUR subdirectory of your TC directory).

Chapter 7, "Debugging in the new IDE," gives you a long tutorial on debugging.

# The IDE

The Turbo C++ IDE contains *windows, menus* (pull-down and pop-up), *dialog boxes,* and a *status line.* Here's a diagram of a typical IDE screen's components:

Figure 3.1
Components of the IDE



F10 To get into the IDE menu system, press *F10,* then press the highlighted letter of any item on the menu bar. (You can also press *Alt* and the highlighted letter; for example, *Alt-F* opens the **File** menu, and so does *F10 F.*)

Esc To get out of the menus, press *Esc* until all menus are closed and you're back at one of the windows.

The Turbo C++ IDE uses *dialog boxes* to display options. A dialog box contains one or more of the items listed in the following table. (If you're itching to try Turbo C++ right now, read the first few paragraphs on page 10.)

Table 3.1
What goes in a dialog box

| Item | What it looks like, what it does |
|---|---|
| **Action Button** | Action buttons are "shadowed" text. If you choose a button (press *Enter* or click), Turbo C++ carries out the related action immediately. |
| **Check box** | A check box is an *On/Off* toggle. Press *Spacebar* or click it to turn the option on or off. When a check box option is turned on, an *x* appears in the brackets: [x]. |
| **Radio button** | Radio buttons are toggles that come in sets of two or more: You can only choose one radio button in a set at a time. With the keyboard, you can move within a set of buttons using the arrow keys. Once you've made your selection, press *Tab* to leave that group with the new button chosen. With the mouse, you can click anywhere on a radio button to choose it. When a radio button is chosen, a bullet appears between the parentheses: (•). |
| **Input box** | An input box prompts you to type in a string (the name of a file, for example). |
| **List box** | A list box contains a list of items from which you can choose (for example, a list of possible files to open). |

Dialog boxes appear when you choose a menu item that is followed by a dialog box icon (...), such as the **F**ile | **O**pen menu item shown in Figure 3.3.

## Full Menus On/Off

Turbo C++ lets you choose between two menu systems, **F**ull Menus On and **F**ull Menus Off. **F**ull Menus Off provides you with a condensed, easy-to-use environment by offering you the minimal command set you'll need to build and debug Turbo C++ programs. **F**ull Menus On provides you with the extra functionality you'll need for more advanced programming tasks. In this tutorial, we use **F**ull Menus set to Off (the default setting).

Figure 3.2 compares the two menu states; as an example, it shows the **C**ompile menu with **F**ull Menus Off and **F**ull Menus On.

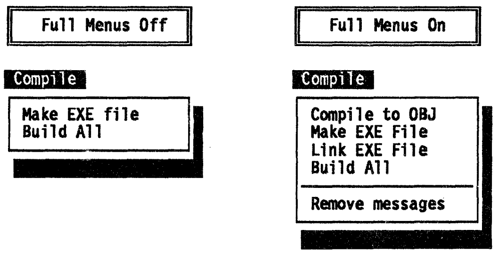


Figure 3.2
Full Menus: Off and On

## Mouse, hot keys, and online help

Before you jump feet first into the IDE, you'll want to know the basics of Turbo C++'s mouse support, hot keys, and online help.

### Mouse

*Refer to your mouse manual if you haven't used your mouse before.*

The Turbo C++ IDE supports a mouse. Use the left mouse button for the IDE. Turbo C++ also lets you redefine your other mouse buttons: Refer to Chapter 1, "The IDE reference," in the *User's Guide* for details. These terms describe mouse actions:

Table 3.2
Mouse talk

| **Action** | **How you do it** |
|---|---|
| **Click** | Press the mouse button and release. |
| **Double click** | Click twice in quick succession. |
| **Drag** | Press the mouse button, move the mouse, release. |
| **Click-drag** | Press the mouse button and release, then press it again, move, and release. |
| **Choose** | Click a menu item, such as a command or dialog-box selector. |

### Hot keys

Many menu and dialog box items have corresponding *hot keys*: one- or two-key shortcuts that immediately activate that command or dialog box. (In a dialog box, unless you're in an input box, you only need to press the highlighted letter to move from one command to another.) The following table lists the most-used Turbo C++ hot keys; see Chapter 1, "The IDE reference," in the *User's Guide* for a more complete list.

Table 3.3
Turbo C++ hot keys

*When you press one of these keys, Turbo C++ carries out that key's function everywhere, except in dialog boxes.*

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

The *status line* at the bottom of the IDE screen lists hot keys you can use in the active window. If you have a mouse, you can click the hot key in the status line instead of pressing the key.

## Online help

Turbo C++, like other Borland products, gives you context-sensitive online help at the touch of a hot key (or the click of a button). You can get help regarding any item in the Turbo C++ IDE, plus help about any Turbo C++ reserved word or library function. In addition, you can copy example programs from the Help window into your own programs.

*You'll use the help system in Lesson 2.*

Turbo C++ also provides a "mini help system" that is always available at a glance. The status line at the bottom of the IDE screen lists any hot keys or commands available in the current window, menu, or dialog box, and provides a brief description of what the current menu or dialog control item does.

## Lesson 1: Starting, loading, and editing

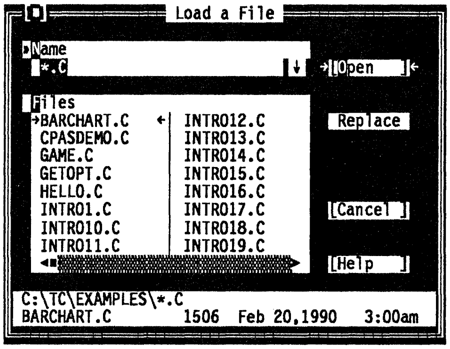
*This lesson should take you 20 minutes to complete.*

Turbo C++'s IDE comes with its own built-in editor. From within Turbo C++, you can open a file, edit, compile, run, debug, and save it to disk again—all with the IDE's easy-to-use features. In this lesson, you'll

- start Turbo C++ at the DOS prompt
- open a file in an Edit window
- copy the file to another Edit window
- use the editor to modify the copied file
- save the modified file to disk

To start Turbo C++, change to the EXAMPLES subdirectory (in your TC directory) and type `..\TC` at the DOS prompt. To load the Turbo C++ program BARCHART.C into the IDE, choose the **F**ile | **O**pen command or press *F3*. Either of these methods displays the Load a File dialog box, which looks like this:

Figure 3.1
The Load a File dialog box



```
BARCHART.C      INTRO12.C
CPASDEMO.C      INTRO13.C
GAME.C          INTRO14.C
GETOPT.C        INTRO15.C
HELLO.C         INTRO16.C
INTRO1.C        INTRO17.C
INTRO10.C       INTRO18.C
INTRO11.C       INTRO19.C

C:\TC\EXAMPLES\*.C
BARCHART.C           1506   Feb 20,1990    3:00am
```

Note that the Open button is the default in this dialog box. If you choose Replace instead of Open, your file will replace any file currently in the active Edit window, instead of placing it in a new window. Leave the Open button selected for this lesson.

From this dialog box, there are two ways to select a file to open:

- You can enter the file name in the Name input box.
- Or, you can choose the file from the Files list.

*If you have a mouse, just double-click the file name in the Files list.*

This time you'll use the Files list.

1. Press *Tab* to activate the Files list.
2. Normally, you would use the arrow keys to highlight your selected file. In this case, BARCHART.C is already highlighted because it's the first file in the list.
3. Press *Enter*.

As soon as you choose the file (press *Enter*), BARCHART.C is displayed in the Edit window. Note the *window number* in the upper right border of the Edit window; this is window 1. The line and column number information for the Edit window is in its lower left corner.

## Creating a new file

One feature of the Turbo C++ IDE is *multiwindowing*—you can have more than one Edit window open at a time. You can open a different or the same file in each window, cut or copy and paste between windows, and move easily from window to window.

Now that BARCHART.C is open in an Edit window, you're ready to start editing the code. But, because you don't want to alter the original program, open a new Edit window with **F**ile | **N**ew.

The new Edit window is NONAME00.C (window 2), which you'll later name MYCHART.C. The double line around the new Edit window tells you that it's the active window.

*To go directly to any Edit window, press Alt and that window's number.*

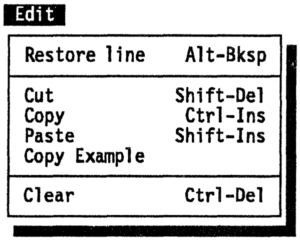
Now go back to the first Edit window (click it or press *Alt-1*). We want you to copy all of BARCHART.C into NONAME00.C. (If you're not familiar with moving around in an Edit window, take a look at the following table or press *F1* while you're in the editor. For a full listing of the editor commands, refer to Chapter 3, "The editor from A to Z," in the *User's Guide*.)

Table 3.1
Moving in an Edit window

| To move | On the keyboard | With the mouse |
|---|---|---|
| A character or line in any direction | Press ↑, ↓, →, or ← | Click the arrow at either end of scroll bar |
| To beginning or end of a line | Press *Home* or *End* | Click at beginning or end of line |
| Up or down a screen | Press *PgUp* or *PgDn* | Click and hold the arrow at either end of the scroll bar to scroll |
| To beginning or end of file | Press *Ctrl-PgUp* or *Ctrl-PgDn* | Click-drag the thumb tab (square icon) at either end of the scroll bar; move to top or bottom of bar; release mouse button |

### Selecting a block

When marking blocks, you can choose between the keyboard and the mouse:



- From the ***keyboard***, place the cursor at the first or last character of the block, then press *Shift* and an arrow key (or *Home*, *End*, *PgDn*, or *PgUp*) to select the block. (You must press *Shift while* you're using the arrow keys.)
- With the ***mouse***, click the first character of the block, then drag to the end of the block. To unselect a block, click anywhere.

A selected block of text displays in reverse video (highlighted). Go ahead and select all the code in BARCHART.C.

### Copying and pasting

Now that you've selected the text in the BARCHART.C window, choose **Edit | Copy** (or press *Ctrl-Ins*). This copies the selected text to a special holding place in memory called the *Clipboard*. Once text is in the Clipboard, you can paste it into a new location—in the same window or a different one.

Move back to NONAME00.C: Press *Alt* and that window's number (or click that Edit window with the mouse). (If you want to move to a particular window but don't know its number, use **Window | List**.)

To paste text from the Clipboard into the Edit window, choose **Edit | Paste** (or press *Shift-Ins*). Next, unselect the pasted block; click anywhere or press *Ctrl-K H*.

Now you're ready to make some changes to the text, and save and name the NONAME00.C file.

## Modifying your new file

BARCHART.C is a program that prompts for ten scores between 0 and 50, computes their percentages, then displays the percentages in a bar chart and prints out the scores and percentages.

In this section, you're going to make some improvements to BARCHART.C:

- change your version of BARCHART.C (NONAME00.C) so that it prompts for five scores between 0 and 35
- use the editor's search-and-replace feature to change a variable name
- copy and paste text from a Help window

*Look for the line number at the bottom of the Edit window.*

Let's start off by editing the file to prompt for a different number of scores within a different range. Here's what you do:

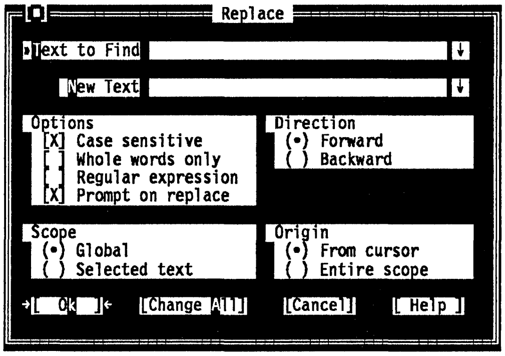
1. Use the cursor keys to move to the end of line 7.
2. Use *Backspace* to remove the number 50, and type in 35.
3. Move to the next line and change the number 10 to 5.

## Searching and replacing

The variable name *i* isn't a very informative identifier, so go ahead and change it to *index*, since that's what the variable is used for. Changing a single-letter variable is a little trickier than it might first appear. You want to change *every* instance of the variable *i*; if you miss one, the program won't compile because you'll have an undefined variable. But you don't want to change every letter *i* in the program, just each instance of the variable *i*.

Fortunately, the **Search** menu includes an option for selective search-and-replace operations. You'll be using the Replace dialog, which looks like this:

Figure 3.2
The Replace dialog


Replace

Text to Find

New Text

Options
[X] Case sensitive
[ ] Whole words only
[ ] Regular expression
[X] Prompt on replace

Direction
(•) Forward
( ) Backward

Scope
(•) Global
( ) Selected text

Origin
(•) From cursor
( ) Entire scope

The Replace dialog box contains two input boxes, three sets of radio buttons, a set of check boxes, and four buttons. You're going to change all instances of variable *i* in NONAME00 to *index*, so go to the beginning of the file (press *Ctrl-PgUp* or use the mouse).

1. Choose **S**earch | **R**eplace to open the Replace dialog box. When you first open the dialog box, the Text to Find input box is active.
2. Type `i`, the name of the variable you want to search for, then press *Tab* to activate the New Text input box.
3. Type `index`, the name you want to substitute for *i*. Press *Tab* again to move control to the Options check boxes.

*To check Whole Words Only, press ↓ to move to it, then press Spacebar. Or, click it.*

Case Sensitive and Prompt to Replace are already checked. That's exactly what you want. But you also need to check Whole Words Only so that the search won't stop at every letter *i* in the program. After you check that option, leave this group as is and press *Tab*, or click, to go to the Direction radio buttons.

4. You want the search direction to be forward, so just go ahead and tab to Scope.
5. Global is already chosen in the Scope set of radio buttons, which means the search will be through the entire document. Now press *Tab* to go to Origin.
6. In Origin, press ↓ to move to Entire Scope (or click it) to choose it.
7. Tab to the Change All button and press *Enter* to initiate the search-and-replace operation (or click the button with the mouse).

At each instance of an *i*, you're prompted whether you want to replace that occurrence. Press *Y* for Yes if the search finds the variable *i*; otherwise press *N* for No.

## Pasting from a Help window

Suppose you want to add this caption to your bar graph:

```
Percentages, Exam 1A
```

The function that outputs text to the screen in graphics mode is **outtextxy**; the help screens describing this function contain code you can copy to the Clipboard and then paste into NONAME00. Here's how to copy **outtextxy**'s example code from the help screen to your source:

1. Choose **I**ndex from the **H**elp menu.
2. Type the word *outtextxy* to go to the entry for **outtextxy**. Then press *Enter* to bring up its help screen.
3. The example is already preselected as a block, so choose **E**dit | Copy **E**xample to copy it into the Clipboard.
4. Press *Esc* to close the Help window.
5. Go to the end of your file and position the cursor at the beginning of the last line.
6. Choose **E**dit | **P**aste to paste the block of code into your file.

You won't use the whole block as it stands. While parts are useful just as they are, others must be modified to suit your purposes, and some are no use to you in this example.

Once you've copied the example into your program, you can modify the lines you want to keep, then discard the rest. For example,

1. You can use the line

   ```
   int midx, midy;
   ```

   as is. Cut this line (from the example text you just added at the end of your program), then paste it into the **makegraph** function's variable declaration section (line 35).
2. Paste these two copied lines into **makegraph** just after the closing brace of the **for** loop (line 49).

   ```
   midx = getmaxx()/2;
   midy = getmaxy()/2;
   ```
3. Modify them to read

   ```
   midx = (getmaxx()/2) - (textwidth("Percentages, Exam 1A")/2);
   midy = getmaxy() - 10;
   ```
4. Paste this copied line into **makegraph** right after the modified *midy* statement:

   ```
   outextxy(midx, midy, "This is a test.")
   ```
5. Modify it to read

   ```
   outtextxy(midx, midy, "Percentages, Exam 1A");
   ```

You need to get rid of the mangled remains of the **outtextxy** sample program at the end of your file. Fortunately, the **E**dit menu provides you with a command for getting rid of blocks of text with just a couple of quick keystrokes.

1. Select the block you want to delete: Start where you typed in **outtextxy**, and stop at the end of the file.
2. Choose **E**dit | **C**lear (or press *Ctrl-Del*).

## Saving your changes

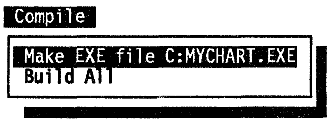
That's it. You've made quite a few changes to your program, so go ahead and save the file to disk. Choose **F**ile | **S**ave (or press *F2*), which brings up the Save Editor File dialog box. At the input box, type in MYCHART.C and press *Enter*.

# Lesson 2: Compiling and running

*This lesson takes 5 minutes.*

In this lesson, you'll

- compile MYCHART.C with the **C**ompile | **M**ake EXE command
- run MYCHART.EXE with the **R**un | **R**un command (so you can see the output)



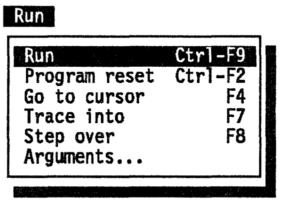
You use the **C**ompile menu to compile the program. (Because the Full Menus option is set to Off for this tutorial, this Compile menu shows only two items. To find out what all the Compile menu options do, see Chapter 1, "The IDE reference," in the *User's Guide*.)

***Note:** If you get error messages stating that Turbo C++ can't find your header files, Turbo C++ probably is not installed with the default subdirectories. See Chapter 1 for details.*

1. Choose **C**ompile | **M**ake EXE File or press *F9* to generate MYCHART.EXE. The Compiling window appears; if there are any errors in your program, it'll tell you here (if so, fix them and recompile). When the program successfully compiles and links, the window displays a flashing message:

   ```
   Success: Press any key
   ```

2. Press any key to return to your program.



To run MYCHART, choose **R**un | **R**un or press *Ctrl-F9*. (The other commands on the **R**un menu are for debugging.) Turbo C++ switches from the IDE to the User Screen, where MYCHART prompts you for input.

Each time MYCHART prompts you for input (five times in all), enter an integer between 0 and 35. As soon as you enter the last number, MYCHART displays all five values and their percentages.

*The User Screen displays both text and graphics output.*

Press *Enter.* Now MYCHART displays the percentages in the form of a bar graph. Notice how nicely the caption is centered under the bar graph. If you're curious about how that happened, check out this line:

```
midx = (getmaxx()/2) - (textwidth("Percentages, Exam 1A")/2);
```

Press *Ctrl-F1* while the cursor is positioned on the function **textwidth** to get language-specific help (or look in Chapter 1, "The run-time library," in the *Library Reference*).

Press *Enter* again to return to the IDE.

## Closing an Edit window

Now you're finished with MYCHART.C. To close its Edit window, choose **Window | Close** (or press *Alt-F3*). Go ahead and close BARCHART.C as well.

If you have a mouse, you can click the close box ([■]) in the upper left corner to close the Edit window.

## Lesson 3: Exiting Turbo C++

When you finish working on a file, the last two things to do are

- save your changes to a file on disk
- exit the Turbo C++ IDE and return to DOS

You've already saved your file, so the last step is to quit Turbo C++ and return to DOS.

- Choose **File | Quit** or press *Alt-X*.

## Where to go for more information

This tutorial was just a quick introduction to the most commonly used parts of the Turbo C++ IDE; many features were touched on briefly or not at all. Remember, you can find in-depth information on every part of the IDE in Chapter 1, "The IDE reference," in the *User's Guide*.

C H A P T E R

4

# An introduction to C

If you've never programmed in C before (or if you have and would just as soon forget the experience), this is the chapter for you. It starts out with simple examples and moves to more complex ones, showing you how to build them into programs. You'll learn how to solve a variety of problems involving numbers, words, and graphics. You are also introduced to some important guidelines for designing and structuring programs. You'll find code for the more complex programs in the EXAMPLES subdirectory. We've included them this way so you can easily load, compile, and run them; this is one of the best ways to learn C.

## A quick history lesson

C was originally developed in the 1970s for use with the UNIX operating system, and virtually grew up with it. When microcomputers with sufficient power came on the market, C compilers were implemented for them. In 1978, Brian W. Kernighan and Dennis M. Ritchie provided the classic definition of C with the first edition of their book *The C Programming Language.* Five years later, the American National Standards Institute (ANSI) began to develop a new standard for the C language. This standard resolves ambiguities in the classic definition and provides new features, including better control over function calls through the use of prototypes. The second edition of Kernighan and Ritchie's book discusses the ANSI standard implementation in detail.

Turbo C++ implements the latest ANSI standard for C. It is also a full implementation of version 2.0 of AT&T's object-oriented version of C called C++. The object-oriented features of C++ are discussed in chapters 5 and 6, "A C++ primer" and "Hands-on C++."

## Basic programming operations

*You can follow along with the examples in this chapter by loading and running the designated programs.*

Computer programs vary greatly in purpose, style, and complexity. Nearly all programs, however, go through a process consisting of three phases:

- describing, collecting, and storing information (data)
- processing the data to achieve the desired result
- displaying and/or storing the results

Any data used by a program has to be described so that Turbo C++ knows how to store and retrieve it. Memory must be set aside to hold the amount of data expected. The program must then use some means to get the actual data into storage—this could involve reading the characters from the keyboard, retrieving data from a file on disk, receiving data over a telephone line, or using some other kind of input device.

Once the data has been stored in numeric variables, character strings, arrays, or more complicated data structures, it must be processed. The processing varies with the purpose of the program, of course: A spreadsheet program might apply a formula to a set of data to calculate a result, while a word-processing program might rearrange lines of text to fit new margins.

Once the data is processed, the results must be made available in some way to the user. Lines of text can be rearranged on the screen or sent to the printer, and the spreadsheet cells can be redisplayed to show their new values. Most data must eventually be stored on disk for later use.

Consider how the foregoing elements of program design are used in this short example program:

*To try out this program, load and run INTRO1.C (**File | Open | INTRO1**). Remember that all these examples are in the EXAMPLES subdirectory.*

```
/* INTRO1.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
      int   bushels;
      float dollars, rate;
      char  inbuf [130];
      printf("How many dollars did you get? ");
      gets(inbuf);
      sscanf(inbuf, "%f", &dollars);
      printf("For how many bushels? ");
      gets(inbuf);
      sscanf(inbuf, "%d", &bushels);
      rate = dollars / bushels;
      printf("You got %f dollars for each bushel\n", rate);

      return 0;
   }
```

The first line of this program, `main()`, defines a *function*, or group of related program instructions. Functions are the building blocks of C programs, much as paragraphs are the building blocks of stories. Every C program has a function called **main**. Most programs have several other functions with appropriate descriptive names. The open brace ({) indicates the beginning of a group of program instructions, or *statements*—in this case, the statements that define what will happen when the function **main** is executed. Each group of statements ends with a closing brace (}).

If functions are like paragraphs, statements are like sentences. Notice that each statement ends in a semicolon (;). While C lets you string several statements together on the same line, we don't recommend this; it makes programs harder to read.

*Describing* The first three statements are

```
int   bushels;
float dollars, rate;
char  inbuf [130];
```

Recall that the first step in writing a program is "describing, collecting, and storing information." In C, you must *declare* each item of data before you can do anything else with it. To declare an item of data, list what *type* of data it is, then give it a name. Here, you have one item that has an **int** type (integer, or whole number), and is named *bushels*. You have a second and third item that both have a **float** type (a floating-point number or a number that has a decimal fraction), namely *dollars* and *rate*. These data items are also called *variables*, since their value can vary according to circumstances. And you have a fourth item, a character *array*, that lets you read and store your user's input. (Arrays are discussed on page 97.)

*Collecting and storing*

The next six statements obtain and store the data we've just described. The **printf** statements prompt for the number of bushels and the number of dollars received for those bushels, and the **gets** and **sscanf** statements get these values and store them in the variables named. Most of the actual work done in a Turbo C++ program is accomplished by calling upon functions provided in the libraries included with Turbo C++. You call a function by giving its name in a statement, along with any information the function requires for processing, enclosed in parentheses. **printf**, **gets**, and **sscanf** are all library functions. They are not actually part of the C language itself. The specifiers **%d** and **%f** in the **sscanf** statements indicate that these data items are to be stored as an integer and a floating-point decimal, respectively. These are called format specifiers; we explain and describe them on page 46.

*Processing*

The next statement, `rate = dollars/bushels`, does the processing part of the program, dividing the number of bushels by the number of dollars to get the dollars per bushel.

*Printing*

The final statement, again using the **printf** function, displays the results of this calculation. Notice that the **printf** statement here specifies a string (message) followed by a comma and the name of the variable *rate*. The **%f** specifies that the value should be formatted as a floating-point value.

When you run the program, the output looks like this:

```
How many dollars did you get? 32
For how many bushels? 24
You got 1.333333 dollars for each bushel
```

## Basic structure of a C program

This next example demonstrates functions, comments, and the preprocessor directives **#include** and **#define**.

*Load and run SALESTAG.C.*

```
/* SALESTAG.C--Example from Chapter 4 of Getting Started.
   SALESTAG.C calculates a sales slip. */

#include <stdio.h>
#define RATE 0.065                         /* Sales Tax Rate */

float tax (float amount);
float purchase, tax_amt, total;

int main()
```

```
{
   char inbuf [130];
   printf("Amount of purchase: ");
   gets(inbuf);
   sscanf(inbuf, "%f", &purchase);

   tax_amt = tax(purchase);
   total = purchase + tax_amt;

   printf("\nPurchase is:  %f", purchase);
   printf("\n         Tax:  %f", tax_amt);
   printf("\n       Total:  %f", total);

   return 0;
}

float tax (float amount)
{
   return(amount * RATE);
}
```

The first line of the program begins with the symbol /*, which begins a *comment.* The symbol */ ends the comment. Turbo C++ ignores all characters between the beginning and end of a comment. You use comments to describe the purpose of a program, function, or statement. Appropriate comments make it easier for you to remember just what a particular part of your program does—and it helps other programmers who may be called on to modify your program.

Lines that begin with the number or pound sign (**#**) are not C language statements, but instructions to Turbo C++ itself. These are called *compiler directives* (or *preprocessor directives*) because they direct the operation of the compiler. The directive `#include <stdio.h>` tells Turbo C++ to read in and compile the contents of the file stdio.h, which is one of the many *header files* (also called *include files*) that Turbo C++ installed for you. These files contain descriptions of the library functions (such as **printf**, **gets**, and **sscanf**), as well as other items that are part of the standard C library. The compiler uses these descriptions to compile the program code for the library functions, along with the code for your program statements. For more information on each of these header files, see Chapter 2, "Run-time library cross-reference," in the *Programmer's Guide.*

*Another way to specify values that won't change is to use the **const** keyword, discussed on page 94.*

The next line, `#define RATE 0.065`, defines a *macro substitution*: It tells Turbo C++ that whenever it sees the word *RATE* in your program code, it is to replace that word with the number 0.065. In a longer program, the tax rate may be referred to many times. If the tax rate changes, you need only change the definition and recompile the program. All of the references will be changed automatically. This is less time-consuming and more accurate than trying to find and change all the references by hand.

The next statement describes a user-defined function **tax**, which is defined later. The description (called a *prototype;* see page 81) says that this function will accept a floating-point number (**float**) and return a result that is also a floating-point number. Putting the description here helps Turbo C++ make sure that your program doesn't try to give the function the wrong kind of data (a character string, for example). The statement

```
float purchase, tax_amt, total;
```

describes three floating-point variables.

The **printf, gets,** and **sscanf** statements prompt for and obtain the amount of the purchase. Now it's time for the actual computing.

```
tax_amt = tax(purchase);
```

calls the user-defined **tax** function. The value of *purchase* (the variable in parentheses) is sent to this function. To find out what the function does, skip down to the bottom of the program, where you see its definition:

```
float tax (float amount)
{
   return(amount * RATE);
}
```

This specifies that the **tax** function takes the value of *amount* that it receives, multiplies it by the defined constant RATE, and returns the result. Thus, when the line

```
tax_amt = tax(purchase);
```

is executed, the tax on the amount of *purchase* is calculated and returned by the **tax** function, then stored in the variable *tax_amt* for later use. In the next line, in fact, this amount is added to *purchase* to obtain *total.*

It may seem unnecessary to have a whole separate function just to calculate the tax, and it is in this program. But it becomes useful in more complicated situations, such as when there are several tax rates to choose from according to the purchaser's county of residence. Perhaps you also have to check a product code to determine whether the item is taxable in the first place. In that case, separating the mechanism for figuring tax makes the main part of the program easier to follow. If necessary, you can later change how the tax is calculated without affecting the rest of the program.

*The output value of purchase doesn't match the input value because of how **float** works; see page 40.*

The final three lines of the main program print out the purchase amount, tax, and total. A sample run looks like this:

```
Amount of purchase: 24.95

Purchase is:  24.950001
        Tax:  1.621750
      Total:  26.571751
```

Clearly some more work will be needed to format your output neatly—dollar amounts should have only two decimal places, and the amounts should be right-justified. That can wait until you learn more about formatting and the **printf** function, however. (If you're concerned about those fractional pennies, Chapter 4, "Memory models, floating point, and overlays," in the *Programmer's Guide* has a brief discussion of using the **bcd** class to get more precise results.)

## Working with numbers

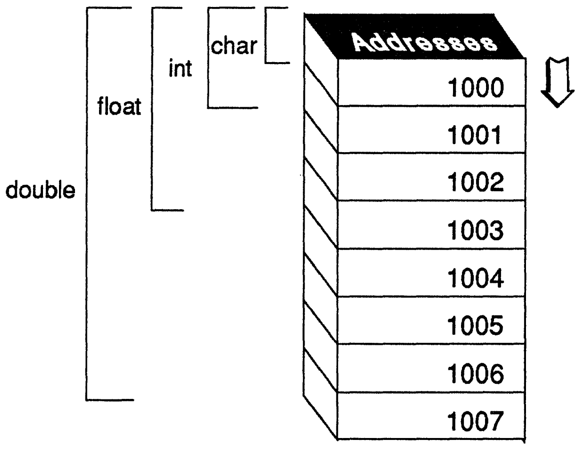
Numbers are the fundamental data used by computers. As you probably know, the actual contents of computer memory consists of binary numbers. These are usually organized in groups of 8 bits (1 byte) or 16 bits (2 bytes, or 1 *word*). Even those computing activities that involve words or graphics basically involve series of numbers stored in memory.

### Numeric data types

The same part of memory could be interpreted as several different kinds of numbers, depending on how many bytes are grouped together. The name of a variable, such as *total*, actually refers to the contents of one or more bytes following a specific address in memory—this address is assigned by Turbo C++ when you first define (give a value to) the variable. But you and the compiler must agree about what kind of number will be represented by a given variable, and thus how many bytes will be stored and fetched starting at the variable's address. You make this agreement by specifying a *data type* when you declare the variable. For example,

```
int   total, count, step;
float cost, profit;
```

Figure 4.1
Interpreting memory locations as numbers (in 1-byte increments)



Each data type represents a different kind of number. You must choose an appropriate type for the kind of number you need. In this variable declaration,

- The variable *total* is of type **int** (integer). When you tell your program to use the value of *total* in a statement, it fetches 2 bytes, starting at *total*'s address.
- The variable *cost* is of type **float** (floating point). When your program uses *cost*, it fetches 4 bytes, starting at *cost*'s address. (This is because a floating-point number needs the two extra bytes to represent the significant digit of the number and the magnitude of the number in terms of powers of two.)

Table 4.1 shows the basic Turbo C++ data types and their variations. Notice the variety of numbers that can be accommodated. This chapter shows you how to use many of these types.

Table 4.1: Data types, sizes, and ranges

| Type | Size (bits) | Range | Sample applications |
|---|---|---|---|
| **unsigned char** | 8 | 0 to 255 | Small numbers and full PC character set |
| **char** | 8 | –128 to 127 | Very small numbers and ASCII characters |
| **enum** | 16 | –32,768 to 32,767 | Ordered sets of values |
| **unsigned int** | 16 | 0 to 65,535 | Larger numbers and loops |
| **short int** | 16 | –32,768 to 32,767 | Counting, small numbers, loop control |
| **int** | 16 | –32,768 to 32,767 | Counting, small numbers, loop control |
| **unsigned long** | 32 | 0 to 4,294,967,295 | Astronomical distances |
| **long** | 32 | –2,147,483,648 to 2,147,483,647 | Large numbers, populations |
| **float** | 32 | $3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$ | Scientific (7-digit precision) |
| **double** | 64 | $1.7 \times 10^{-308}$ to $1.7 \times 10^{308}$ | Scientific (15-digit precision) |
| **long double** | 80 | $3.4 \times 10^{-4932}$ to $1.1 \times 10^{4932}$ | Financial (19-digit precision) |
| **near pointer** | 16 | Not applicable | Manipulating memory addresses |
| **far pointer** | 32 | Not applicable | Manipulating addresses outside current segment |

## Integers

The basic integer type is **int**, which can express either negative or positive numbers, but within a limited range (-32,768 to 32,767).

Here's an example program that performs some operations with integers:

```
#include <stdio.h>

main()

{
   int bags, pounds;
   int total;
   pounds = 50;
   bags = 1000;
   total = bags * pounds;
   printf("There are %d lbs. in 1000 bags of beans\n", total);
   return 0;
}
```

The output from this program is a little surprising:

```
There are -15536 lbs. in 1000 bags of beans
```

*The compiler doesn't warn you if you try to store a value that doesn't fit into the specified data type.*

The total of *bags* * *pounds*, 50,000, is too large for an ordinary **int** (which you can verify from Table 4.1). When your program tried to store 50,000 in a type that could only hold 32,767, the result overflowed. How do you solve this problem? Use **long int.**

### The long modifier

**long int**, which is usually abbreviated just as **long**, gives you a larger integer range. You can solve the problem of the negative beans by declaring:

```
long total;
```

*You'll also need to change the **%d** in the **printf** statement to **%ld**. We explain why on page 51.*

This gives you room for more pounds of beans than you'd ever be likely to see, because a **long**, which is stored in 32 bits instead of the 16 used by an ordinary **int**, can accommodate a value between –2,147,483,648 and 2,147,483,647. But what about *pounds*? This variable should be fine as an **int**, since the weight of one bag is unlikely to exceed 32,767 pounds. The variable *bags*, however, might conceivably exceed 32,767, so make it a **long** also. Why not use **long** instead of **int** variables for everything? A **long** takes up 4 bytes of memory, while an **int** takes only 2. If you have many variables, you'll end up wasting a lot of memory.

```
int pounds;
long bags, total;
```

## Signed and unsigned variables

*To decide which data type to use, consider the possible results of a calculation or other operation.*

All the data types listed in Table 4.1 are ***signed*** by default—one of the bits in the stored value is used to indicate whether the number is positive or negative. (Those marked **unsigned** are, of course, explicitly **unsigned.**) Some values encountered in your work can be either positive or negative—for example, temperatures and bank balances. Many other values, however, are never negative—a business can't have a negative number of employees, for example. By adding the word **unsigned** to any data type, you restrict its range to positive numbers. Since a sign bit is no longer needed, this doubles the maximum value stored by the type. For example, while an ordinary (signed) **int** ranges from –32,768 to 32,767, an **unsigned int** ranges from 0 to 65,535. (An **unsigned long** ranges between 0 and 4,294,967,295.) The preceding program example would also have worked correctly if you had declared

*Since the default for all numeric data types is signed, you don't have to declare **signed int** or **signed long**.*

```
unsigned int total;
```

though you'd be getting uncomfortably close to the limits of the **unsigned int** type.

## Floating-point numbers

Many numbers involve a fractional part set off with a decimal point, such as prices in dollars and cents. These are called *floating-point numbers* (also often called *real* numbers). Most exact measurements involve fractions: If you buy screws at the hardware store, you'll probably have to specify the diameter in fractions of an inch. The **float** data type covers such situations. Here's an example:

*To try out this program, load and run INTRO2.C.*

```
/* INTRO2.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char  inbuf[130];
   float num, denom;      /* numerator and denominator of fraction */
   float value;           /* value of fraction as decimal */

   printf("Convert a fraction to a decimal\n");
   printf("Numerator: ");
   gets(inbuf);
   sscanf(inbuf, "%f", &num);
   printf("Denominator: ");
   gets(inbuf);
   sscanf(inbuf, "%f", &denom);

   value = num / denom;  /* convert fraction to decimal */

   printf("\n %f / %f  = %f", num, denom, value);

   return 0;
}
```

The program prompts for the numerator and denominator of a fraction, then converts them to a decimal value and prints the result. For example,

```
Convert a fraction to a decimal
Numerator: 7
Denominator: 8

 7.000000 / 8.000000  = 0.875000
```

Clearly *value* must be a **float** in order to hold a fraction, but you may not realize that either *num* or *denom* has to be of the **float** type if you wish to divide *num/denom* correctly. Try this example:

*If you divide integers by integers, the result is rounded down to the nearest whole number before it is assigned to the float variable.*

```
#include<stdio.h>

int main()

{
   int num = 3, denom = 4;
   float value;
   value = num / denom;
   printf("\n%f", value);
   return 0;
}
```

The result is a big fat zero—or more precisely, 0.000000, not the 0.75 you'd expect.

### Double and long double

The **double** and **long double** types are like **float**, only they accommodate larger numbers with more precision. Precision is important in both scientific and financial calculations. You might want to use your system to print checks for large sums of money, such as $125,375,750.75.

```
#include <stdio.h>

int main()
{
   float amount = 125375750.75;
   printf("\nPay the sum of %f dollars\n", amount);
   return 0;
}
```

When the check is printed out, you'll see

```
Pay the sum of 125375752.000000
```

The result is an overpayment of $1.25. That may not seem like much, but people expect computers to be 100% accurate, particularly where money is involved. This error occurred because the **float** type is limited to seven digits of precision, and the value assigned to *amount* has eleven digits, ten nonzero. Change *amount* to a **double** and run the program again; you'll find that the amount is now correct.

## Variables

As you have learned, every variable must be declared before it can be used. A *declaration* consists of a data type followed by one or more variable names. Declarations simply tell Turbo C++ that you intend to use a particular variable, and what type of data it will store.

```
int hours;
float total_pay, pay_rate;
long id_number;
```

### Initializing variables

You also need to *initialize* a variable—set it to a specific value, such as 0. What do you think the following program will display?

```
#include <stdio.h>

int main()
{
   int something;
   printf("%d", something);
   return 0;
}
```

The result will vary—on our machine, it was –32,417. Did you notice that the program did not assign any value to the variable *something* before trying to print it out? With the exception of global or static variables (discussed later), variables in C do not have a default value. (Some languages, such as BASIC, typically give numeric variables a default value of 0.) The value of *something*, therefore, is whatever number happens to be stored at the address Turbo C++ assigned to the variable. This value is unpredictable. In fact, if you compiled this program, you might have noticed a warning in the Message window: "Possible use of 'something' before definition in function **main**." When you get this warning, you should check the variable named to make sure you initialize it before you use it for anything.

### Assignment statements

You give a value to a variable with an *assignment statement*. Assignment consists of a variable name followed by an equals sign and the value to be assigned. Here are some examples:

```
count = 0;
total = purchase + tax_amt;
tax_amt = tax(purchase);
```

In the first statement, an actual number, or *numeric constant,* is assigned to the variable *count.* The second statement uses an expression to assign the sum of *purchase* and *tax_amt* to the variable *total.* An *expression* is any combination of values and *operators* (such as **+** or *****) that yield a single value. In C, you can use an entire expression just about anywhere that you can use a single numeric value. You can assign it to a variable, send it to a function for processing, or print it with **printf**.

The third statement is slightly more complex: It first calls the function **tax**, giving it the value of the variable *purchase.* The function uses this value and other information to calculate the tax. The function returns a value to the calling statement: In other words, the function call **tax**(*purchase*) is replaced by an actual value, such as 1.14. The assignment statement then assigns this value to *tax_amt.* Assignment statements using function calls are very common in C.

## Combination assignments

C often lets you combine two or more distinct operations in a single statement. You can declare a variable and assign it a value in a single statement. Instead of

```
float total_expenses;
total_expenses = 0;
```

most C programmers write

```
float total_expenses = 0;
```

You can also assign several variables the same value in one statement. A word processor might start processing text by setting

```
page = line = column = 1;
```

This works because an assignment statement not only assigns a value, it also provides a value that can be used by other parts of a statement in which it is embedded. That is, `column = 1` assigns 1 to *column,* and makes this value, 1, available. Moving right to left, we get the equivalent of `line = 1`. In turn, that assignment passes on the value 1, so the final assignment is `page = 1`.

But don't go overboard. It is often better to declare and initialize one variable per statement, so you can include a comment describing the purpose of each variable:

```
int lines = 0;     /* Lines of text, ending in new line char */
int words = 0;     /* Words are groups of characters surrounded */
                   /* by space, tabs, or new lines */
int chars = 0;     /* Every character is counted */
```

Taking the time to do this might also alert you to potential problems. For example, is it really a good idea for *chars* to be an **int**?

## Naming names

It's time now to consider what names you can give to variables. C is quite flexible in this regard. User-supplied names (called *identifiers*) must follow these rules:

- All identifiers must start with a letter (*a* to *z* or *A* to Z) or an underscore (_).
- The rest of the identifier can use letters, underscores, or digits (0 to 9). Other characters (such as punctuation marks or control characters) cannot be used.
- The first 32 characters of an identifier are significant. This means that

  *C++ identifiers are significant to any length.*

  *The_total_amount_of_money_in_my_checking_account*

  and

  *The_total_amount_of_money_in_my_charge_account*

  would be considered by Turbo C++ to be the same variable. Of course it would be awkward to use such names, anyway.
- Identifiers are case-sensitive. This means that *amount* and *Amount* are completely separate variables.

By these rules, *deduction*, *tax_status*, and *amt_1099* are all legal identifiers, while *1989_tax* and *stop!* are not. (*1989_tax* begins with a digit instead of a letter or underscore, and *stop!* contains an exclamation point, which is not a letter, underscore, or digit.)

Besides following the rules, it is important to give some thought to naming your variables. Here are some suggestions:

- The name should describe what the variable contains. *a* doesn't tell you anything. *amt* is better, but *taxable_amount* is most clear and specific.

- Use capital letters or underscores to separate words in a long identifier. *PricePer100* or *price_per_100* are much easier to read than *priceper100*.
- Use comments to describe the nature and purpose of a variable, particularly if it is not obvious.

## Fielding an input value: sscanf

You have already seen examples of the use of the **gets** and **sscanf** functions for obtaining data from the keyboard. You know that **sscanf** stores data in specified variables, and you've seen it used with different numeric data types (**int** and **float**, for example). Since different data types are stored differently in memory, you need a way to tell **sscanf** what kind of data you wish to store. So let's look at the anatomy of **sscanf** more closely. Here is the syntax of a call to **sscanf**:

**sscanf**(*buffer, "format string", [address, address, ...]*)

Compare this to an actual **sscanf** statement you've seen before:

```
sscanf(inbuf, "%f", &num);
```

*buffer* is the input array where your data was temporarily stored by **gets**. *format string* contains one or more *format specifiers*. Format specifiers consist of a percent sign (%) followed by a letter indicating the type of data to be stored. The group of one or more format specifiers is placed between double quotes. For example, **%d** specifies an **int** value, **%f**, as in the example just given, expects a **float**, and **%s** indicates a string of characters. The format string **%c %d** asks for both a single character and an **int** value. The most commonly used formats are shown in Table 4.2.

Table 4.2
sscanf and printf format specifiers

*This table also shows **printf** format specifiers, discussed on page 49. Note that case is important with most format specifiers.*

| Function | sscanf | printf |
|---|---|---|
| ***Size of value (modifies data type):*** | | |
| Specify short integer | **%hd** | |
| Specify long integer | **%ld** | **%ld** |
| Specify double | **%lf** | **%lf** |
| Use with float to indicate a long double | **%Lf** | **%Lf** |
| ***Type of data to be read or displayed:*** | | |
| Single character | **%c** | **%c** |
| Signed integer | **%d** | **%d** |
| Signed double or float in exponential format | **%e** | **%e** |
| Signed double or float in decimal format | **%f** | **%f** |
| Character string | **%s** | **%s** |
| Unsigned decimal integer | **%u** | **%u** |

The other necessary part of an **sscanf** call is the *address* at which the data is to be stored. Every variable in C has a specific address in memory. Most of the time you don't need to worry about addresses—you just name the variable to get its value. For example, `count + 1` evaluates to the current value of the variable *count* plus one. In the case of **sscanf**, however, you don't need the value of the variable, you need its address. The address operator **&** (the ampersand) is used with a value to get that address. This program shows the difference between a variable's address and the value stored there:

*To try out this code, load and run INTRO3.C.*

```
/* INTRO3.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char inbuf[130];
   int  number = 10;
   printf("Address of variable number = %ld\n", &number);
   printf("Value stored at variable number = %d\n", number);
   printf("Enter a new value for the variable: ");
   gets(inbuf);
   sscanf(inbuf, "%d", &number);
   printf("New value stored at variable number = %d\n", number);

   return 0;
}
```

The output looks like this (the address may vary):

```
Address of variable number = 65498
Value stored at variable number = 10
Enter a new value for the variable: 33
New value stored at variable number = 33
```

The following program illustrates more variations on **sscanf**:

*To try out this code, load and run INTRO4.C. Note how we precede calls to **sscanf** with a call to **printf** or **puts** that describes the value wanted.*

```
/* INTRO4.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char inbuf[130];
   long transaction_number;
   int cashier_number;
   char transaction_code;
   float purchase_amount;

   printf("Enter transaction number: ");
   gets(inbuf);
   sscanf(inbuf, "%8ld", &transaction_number);

   printf("\nEnter your cashier number: ");
   gets(inbuf);
   sscanf(inbuf, "%2d", &cashier_number);

   printf("\nEnter transaction type code: ");
   gets(inbuf);
   sscanf(inbuf, "%c", &transaction_code);

   printf("\nEnter amount of purchase: ");
   gets(inbuf);
   sscanf(inbuf, "%f", &purchase_amount);

   return 0;
}
```

The first **sscanf** statement will read a **long** (**long int**) of not more than eight digits, while the second **sscanf** statement will read an **int** of not more than two digits. (In a real application, you would have to perform further error checking. For one thing, if you type in more digits than are specified, **sscanf** simply starts storing them in the next variable specified, or ignores them if there are no more variables, as in this example.)

You can ask for the values of more than one variable in the same call to **sscanf**; for example,

```
gets(inbuf);
sscanf(inbuf, "%8ld %2d", &transaction_number, &cashier_number);
```

combines the first two **sscanf** calls in the preceding example. Since only one call to **gets** is made, only one line of input is read.

By default, **sscanf** assumes the user will separate numeric variables with spaces. You can include other separator characters in the format string; in which case, you *must* type in each value exactly as entered:

```
#include <stdio.h>

int main()
{
   char inbuf[130];
   int  hours, minutes, seconds;
   printf("Enter new time in hh:mm:ss ");
   gets(inbuf);
   sscanf(inbuf, "%d:%d:%d", &hours, &minutes, &seconds);
   return 0;
}
```

## Displaying a variable's value

The counterpart to **sscanf**, which gets a value for a variable, is **printf**, which displays the value of a variable onscreen. A call to **printf** consists of the following:

**printf**(*"format string", item, item, ...*)

*format string* can contain optional text to be displayed. If you're displaying the values of one or more items (variables, expressions, constants, and so on), you must include a *conversion specification* for each item. These conversion specifications are analogs of the format specifiers discussed with **sscanf** earlier, but there are additional features for formatting the output. The specifier consists of a percent sign followed by a symbol for the type of data involved.

Earlier you used **printf** statements to display a purchase amount, tax, and total:

```
printf("\nPurchase is: %f", purchase);
printf("\n       Tax: %f", tax_amt);
printf("\n     Total: %f", total);
```

The results were displayed like this:

```
Purchase is: 24.950001
        Tax: 1.621750
      Total: 26.571751
```

*The number of digits includes the decimal point. %6.2f, for example, prints five digits, two of which are to the right of the decimal point.*

How can you set up the conversion specifier to make Turbo C++ display these values properly aligned with only two decimal places (which is appropriate for dollar amounts)? Here's one solution: After the **%** sign, put a number to indicate total digits you want displayed, followed by a period and the number of decimal places you want.

```
printf("\nPurchase is: $%6.2f", purchase);
printf("\n       Tax: $%6.2f", tax_amt);
printf("\n     Total: $%6.2f", total);
```

This displays the values as

```
Purchase is: $ 24.95
        Tax: $  1.62
      Total: $ 26.57
```

The next example shows how **printf** can display several types of numbers in different formats:

*To try out this program, load and run INTRO5.C.*

```
/* INTRO5.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int   int_num = 999;
   float float_num = 99.99895;
   long  double big_num = 1250500750.75;

   printf("12345678901234567890\n");
   printf("%d\n", int_num);
   printf("%6d\n", int_num);
   printf("%f\n", float_num);
   printf("%6.3f\n", float_num);
   printf("%e\n", big_num);
   printf("%Le\n", big_num);

   return 0;
}
```

Here is the output:

```
12345678901234567890
999
   999
99.998947
99.999
-3.55361e-207
1.250501e+09
```

The first **printf** statement prints 20 digits as a ruler to show how the other numbers line up. It also illustrates that **printf** doesn't have to print the value of a variable or expression—it can print just a text string if you desire. The next statement prints the value of *int_num* (999) as is. The third statement specifies a 6-digit field; since the default is right-justification, the number 999 is displayed three spaces over from the left. (If you specify a field width too small for the number of whole digits in a number, the specification will be overridden and the full number of digits displayed. This is sensible: It means that 999 won't be displayed as 99 just because you made an error and specified **%2d.**)

The next two statements display the value of *float_num*. In the first statement, the value isn't exactly correct: 99.998947 is displayed, rather than 99.99895. This is because the **float** type declared for *float_num* guarantees seven digits of precision, but **printf** by default tries to print as many digits as it is given to the left of the decimal point, plus six digits to the right of the decimal point, even if some of the digits end up inaccurate. For many applications, this small difference may not matter, but it is important to realize that when you have exceeded the precision for the data type involved, having **printf** show more digits buys you no accuracy and may even mislead the user.

To avoid the display of spurious digits, use a precision specifier to specify how many digits you want after the decimal point. The next statement specifies **%6.3f**, which includes a maximum field width of six and a decimal precision of three places. When the precision is less than the number of digits available, the last digit is rounded up or down as appropriate.

## Type conversion in printf

*Do not specify more digits than the data type can accurately hold.*

**printf** uses the conversion specifier to try to convert the value to whatever type is specified, regardless of the actual data type of the value. For example, if you have a **float** variable *dollars* and use the specifier **%d**, the value 24.95 will end up as 0 or some seemingly random number because **printf** will read 2 bytes (an **int**) rather than 4 (**float**) starting at the address of *dollars*. On the other hand, if you declare a variable as an **int**, then try to display it with **%f**, you will also get a wrong result because **printf** will try to read 4 bytes from a 2-byte variable.

**printf** is a complicated function with a large amount of code for handling the various formats. Turbo C++ links in the part of the code that handles floating-point values only if you use floating point in your program. If your program never uses a floating-point type, and you use the **%f** specifier with an **int**, you will get a run-time error message telling you that the floating-point formats were not linked. A more compact and readable alternative to **printf** and **sscanf** is provided by the C++ streams library, discussed in Chapter 5.

The preceding example uses the **%e** specifier to display *big_num*, which was declared to be of type **long double**. Unfortunately, the specifier tries to print this as an ordinary **double**, again producing a nonsense value with a huge negative exponent. The last statement uses the specifier **%Lf**, which correctly specifies **long double**.

## Formatting with escape\ sequences

There are a number of characters that control how text appears onscreen; for example, the tab character advances the cursor to the next tab position, the newline character moves the cursor to the next line, and the formfeed starts a new screen or page of text. **printf** lets you include any of these characters (and others) in the text to be printed, simply by prefixing the symbol for the character with a backslash (\). The backslash is called an *escape* because it tells Turbo C++ to interpret the following character not as a literal *n* or *f* or whatever, but as the symbol for a special character.

Indeed, you have already seen numerous examples using **\n** in a string being displayed with **printf**. While the print statement in languages such as BASIC automatically advances the cursor or print head to the next line, there is no such default in C. This gives you more flexibility, since you can use separate **printf** statements to display text on the same line, and advance to the next line only when you specifically wish to. The next table lists Turbo C++'s escape sequences.

Table 4.3
Character escape sequences

*The octal and hexadecimal escape sequences are different from Turbo C 2.0. See Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide for details.*

| Sequence | Name | Meaning |
|---|---|---|
| \a | Alert | Sounds a beep |
| \b | Backspace | Backs up one character |
| \f | Formfeed | Starts a new screen or page |
| \n | Newline | Moves to beginning of next line |
| \r | Carriage return | Moves to beginning of current line |
| \t | Horizontal tab | Moves to next tab position |
| \v | Vertical tab | Moves down a fixed amount |
| \\ | Backslash | Displays an actual backslash |
| \' | Single quote | Displays an actual single quote |
| \" | Double quote | Displays an actual double quote |
| \? | Question mark | Displays an actual question mark |
| \OOO | | Displays a character whose ASCII code is an octal value (one to three digits) |
| \xHHH | | Displays a character whose ASCII code is a hexadecimal value (one or more digits) |

- "Newline" on MS-DOS systems is equivalent to a carriage return (CR) plus a linefeed (LF). This is not true of some other systems.
- A backslash in front of the single and double quotes is needed only when Turbo C++ would otherwise interpret these characters as having a special meaning. For example, `"` normally delimits a string. To print a string in quotes, use `"\"a string in quotes\""`.
- The octal or hexadecimal values are often used to send special graphics characters or printer control characters. For example, `printf("\xDB")` on the IBM PC displays a solid square character.

After **\n**, the most commonly used escape sequence is probably **\t**, the tab character. It is useful for aligning tables of numbers. For example, this code

*To try this out, load and run INTRO6.C.*

```
/* INTRO6.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int i = 101, j = 59, k = 0;
   int m = 70,  n = 85, p = 5;
   int q = 39, r = 110, s = 11;

   printf("\tWon\tLost\tTied\n\n");
   printf("Eagles\t%3d\t%3d\t%3d\n", i, j, k);
   printf("Lions\t%3d\t%3d\t%3d\n", m, n, p);
   printf("Wombats\t%3d\t%3d\t%3d\n",  q, r, s);

   return 0;
}
```

produces the following neatly formatted table:

```
        Won     Lost    Tied

Eagles  101      59       0
Lions    70      85       5
Wombats  39     110      11
```

## Arithmetic operators

Now that you know how to get and display values for different kinds of variables, let's look more closely at the variety of operators provided by Turbo C++. You have already seen several operators: the assignment operator (**=**) and four arithmetic operators (**+**, **–**, *****, and **/**, for addition, subtraction, multiplication, and division, respectively).

These operators work pretty much the way you would expect them to, though with some differences: For example, dividing two **int** values gives you an **int** result, with any fraction dropped. There is also a specific order, called *precedence*, in which operators take effect. For arithmetic operators, multiplication and division come before addition and subtraction. Try to guess the four numbers that will be displayed by this program (to try it out, load and run INTRO7.C):

```
/* INTRO7.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   float result;
   result = 1.0 + 2.0 * 3.0 / 4.0;
   printf("%f\n", result);
   result = 1.0 / 2.0 + 3.0;
   printf("%f\n", result);
   result = (1.0 + 2.0) / 3.0;
   printf("%f\n", result);
   result = (1.0 + 2.0 / 3.0) + 4.0;
   printf("%f\n", result);

   return 0;
}
```

*It doesn't hurt to use parentheses, even if they aren't strictly needed. They can make expressions easier to read.*

Here they are. How did you do?

```
2.500000
3.500000
1.000000
5.666667
```

In the first expression, the multiplication *2.0 * 3.0* is done first, yielding 6. Next, *6 / 4.0* gives 1.5, which is finally added to 1.0 to get the final result, 2.5. Notice that when operators have equal precedence (* and / have equal precedence, as do + and –), operations are done from left to right.

In the second expression, the division is done first, then the addition, so the result is *0.5 + 3*, or 3.5.

In the third expression, (*1.0 + 2.0*) is in parentheses, so it is performed first: The result, 3.0, is then divided by 3 to get 1.

Finally, the last expression places *1.0 + 2.0 / 3.0* within parentheses. Within the parentheses, the usual rules are followed: 2 is divided by 3, and then added to 1. The result, 1.666667, is then added to 4 to get 5.666667.

*modulus (%)*

The *modulus* operator (%) divides two numbers and keeps only the remainder. For example, the expression *5 % 2* gives a result of 1, while *18 % 3* gives 0, since 3 divides evenly into 18.

## Arithmetic and type conversion

What happens if you add an **int** to a **float**? You would want the result to be a **float** so that any fractional part is retained, and that is what happens. Turbo C++ *promotes* smaller types to larger ones according to a set of rules (see the next table). From the table, you can see that when an **int** and a **float** are added, the **int** is promoted to a **float**. The two numbers are then added, resulting in a **float**.

Table 4.4
Type promotions for arithmetic

*Some types in this table have not been discussed yet.*

| Type | Converts to |
|---|---|
| These types are converted automatically: | |
| **char** | **int** |
| **unsigned char** | **int** |
| **signed char** | **int** |
| **short** | **int**[1] |
| **unsigned short** | **unsigned int**[1] |
| **enum** | **int** |
| **float** | **double** |
| Then these rules are applied, until both operands have the same type: | |
| *If either operand is...* | *The other is converted to...* |
| **long double** | **long double** |
| **double** | **double** |
| **float** | **float** |
| **unsigned long** | **unsigned long** |
| **long** | **long** |
| **unsigned** | **unsigned** |

[1]This is an ANSI requirement. However, **short** and **int** are the same size for all C compilers on the PC, so no conversion is done.

## Typecasting

It is sometimes useful or necessary to explicitly convert a data item to a specified type. For example, if you have

```
#include <stdio.h>

int main()
{
   int a = 5, b = 2;
   printf("%d", a / b);
   return 0;
}
```

you'll get a result of 2, since integer division drops any fractional part. If, however, you do it this way:

```
#include <stdio.h>

int main()
{
   int a = 5, b = 2;
   printf("%f", (float) a / (float) b);
   return 0;
}
```

the values *a* and *b* will be converted to the type enclosed in parentheses (**float** in this case) before the division, so the value of the expression will be 2.5. This forced conversion is called a *type cast*, or just a *cast*.

## Combining arithmetic and assignment

A common operation in programming involves adding a fixed amount to a variable (*incrementing* it). For example, if a program is counting words, when it finds a word, it will do something like *total_words* = *total_words* + *1*. Later, you will also see how loops usually involve repeatedly adding or subtracting a number until a variable reaches a specified limit.

A shorthand way of doing things in C is to perform arithmetic and assignment in one step. You can combine any binary arithmetic operator with the assignment operator. The preceding statement can also be written as `total_words += 1`. Read this as "add 1 to the current value of *total_words* and assign this quantity as the new value of *total_words*." Similarly, a checkbook-balancing program might execute the statement `balance -= check_amt` (subtract the amount of the check from the balance and make that the new value of *balance*). The somewhat less common combinations ***=** and **/=** work in the same way.

## Increment and decrement

Adding and subtracting exactly one is so common an operation that two special operators, increment (**++**) and decrement (**– –**), are provided for the purpose. Thus `++total_words` does exactly the same thing as `total_words += 1`. A program that does a countdown for the space shuttle might use `count--` in a loop until zero is reached.

The increment and decrement operators can come either before or after the affected variable. When the operator comes before the variable it is applied to the variable *first,* and then the result is used in the expression as a whole. When the operator comes after the variable, the value of the variable is used first, and *then* the operator is applied to the variable. For example,

```
#include <stdio.h>

int main()
{
   int val = 1;

   printf("val is %d and then post-incremented\n", val++);
   printf("val is now %d\n", val);
   printf("val is pre-incremented to %d\n", ++val);
   return 0;
}
```

which gives the results

```
val is 1 and then post-incremented
val is now 2
val is pre-incremented to 3
```

In the first **printf** statement, *val* is still 1 when it is printed, but becomes 2 afterward, as shown in the second **printf**. In the third statement, *val* is incremented first, so it is 3 by the time it is displayed by **printf**.

## Working bit by bit

Sometimes you'll find that you have to manipulate the actual bits that make up each byte in memory. C provides a set of bitwise operators, shown in the next table.

Table 4.5
Bit manipulation operators

| Operator | Meaning |
|---|---|
| Operators that take two operands: | |
| **&** | AND; if both bits are 1, result is 1. |
| **\|** | OR; if either bit is 1, result is 1. |
| **^** | Exclusive OR; if only 1 bit is 1, result is 1. |
| **>>** | For a **signed int**, shift bits right the number of times specified by following number; fill in 1s at left if the number is negative, 0s if positive. For an **unsigned int,** fill in 0s to left. |
| **<<** | Shift bits left the number of times specified by following number; fill in zeros at right. |
| Operator that has a single operand (unary operator): | |
| **~** | 1's complement; reverse all bit values. |

The following program illustrates the use of these operators:

```
#include <stdio.h>

int main()
{
   printf("1 & 1 is %d\n", 1 & 1);
   printf("1 | 1 is %d\n", 1 | 1);
   printf("1 ^ 1 is %d\n", 1 ^ 1);
   printf("255 << 2 is %d\n", 255 << 2);
   printf("255 >> 2 is %d\n", 255 >> 2);
   printf("~255 is %u\n", ~255);
   return 0;
}
```

Remember that the integer 1 is stored in 2 bytes as 00000000 00000001, while 255 is 00000000 11111111. Here is the output:

```
1 & 1 is 1
1 | 1 is 1
1 ^ 1 is 0
255 << 2 is 1020
255 >> 2 is 63
~255 is 65280
```

Notice that the **&** operator gives 1 only if the corresponding bits are both 1. This makes it useful for masking off selected bits of a value by using a bit pattern that has zeros in the positions you wish to turn off. The | operator turns on a bit if either or both values have a 1 in that position. If you want to guarantee that a certain bit is on, "OR" that value with a value that has a 1 in that position.

The fourth statement left-shifts the value 255 twice (00000000 1111111 becomes 00000011 11111100). Since each place in a binary number is two times the preceding place, this is equivalent to multiplying *255 * 2 * 2*, or 1020. In the next statement, 255 is right-shifted twice, so 00000000 1111111 becomes 00000000 00111111, or 63. Finally, the last statement turns 00000000 11111111 into 11111111 00000000. This is equivalent to 65,535 – 255, or 65,280.

## Expressions

You have now seen a variety of statements that use expressions, so this is a good time to review how expressions work in general. Remember that an expression is any combination of variables, defined constants, or literal numbers which together with one or more operators yield a single value and possibly produce one or more side effects. Thus

```
purchase * TAX_RATE
dollars / bushels
count++
STATUS & SWITCH_ON
```

are all examples of expressions. An expression can be assigned to a variable with the assignment operator (=). It can be displayed by **printf** in the same way as a single variable. The preceding program, for example, used statements such as

```
printf("255 << 2 is %d\n", 255 << 2);
```

### Evaluating an expression

Turbo C++ evaluates expressions by applying the operators involved in order of their precedence, starting first with any parts of the expression that are enclosed in parentheses. Table 4.6 lists all of the C operators in order of their precedence and associativity. *Associativity* is the direction in which the compiler evaluates the operators and operands. For example, the various assignment operators (**=**, **+=**, ***=**, and so on) associate from right to left (assigning the expression on the right to the variable on the left), while the arithmetic operators (*****, **/**, **+**, **–**, and **%**) associate from left to right.

*Precedence* is the order in which evaluations are done. For example, multiplication is done before addition, so that the statement

```
count = 5 + 3 * 4
```

results in 17, not 32.

The best way to become familiar with these rules is to use an expression with a **printf** statement so you can see the result, then check the table to see how that result was arrived at. (As an alternative, Chapter 7, "Debugging in the new IDE," shows how you can use the built-in Turbo C++ debugger to evaluate many kinds of expressions.)

The rules of precedence and associativity are as follows:

- Unary operators (operators with only one operand, such as **++**) have a higher order of precedence than binary operators (such as **/**).
- Arithmetic operators come ahead of comparison operators.
- Greater than and less than come ahead of equals and not-equals.
- Comparison operators come ahead of bit manipulation operators (except for left and right shifts).
- Bit manipulation operators come ahead of logical operators.
- Logical AND (**&&**) comes ahead of logical OR (**||**).
- Everything except for the comma operator comes before assignment operators.

Table 4.6
Precedence and associativity of operators

*Operators, precedence, and associativity are explained in full in Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide.*

| **Operators** | **Associativity** |
|---|---|
| **( ) [ ] –> :: .** | Left to right |
| **! ~ + – ++ – – & *** (*typecast*) **sizeof new delete** | Right to left |
| **.* –>*** | Left to right |
| *** / %** | Left to right |
| **+ –** | Left to right |
| **<< >>** | Left to right |
| **< <= > >=** | Left to right |
| **== !=** | Left to right |
| **&** | Left to right |
| **^** | Left to right |
| **\|** | Left to right |
| **&&** | Left to right |
| **\|\|** | Left to right |
| **?:** (conditional expression) | Right to left |
| **= *= /= %= += –= &= ^= \|= <<= >>=** | Right to left |
| **,** | Left to right |

## Assigning a value in an expression

You have been seeing examples of the assignment operator used in a complete statement, such as the initialization of a variable, `total = 0`. Assignment statements like `total = count = line = 0` remind us that every assignment is also an expression. What do you think this program will display?

```
#include <stdio.h>

int main()
{
   int val;
   printf("%d\n", val = 7);
   return 0;
}
```

If you guessed 7, you're right. The value of an assignment statement as an expression is the value assigned.

You have also seen that a call to a function that returns a value has that value. For example, `total += tax(total)` is equivalent to

```
tax_amt = tax(total);
total = total + tax_amt;
```

The first form eliminates the extra variable at the cost of being a bit more cryptic.

# Characters and strings

There's more to life than numbers. It's time to look at characters and character strings. As you probably know, a character—whether an uppercase or lowercase letter of the alphabet, a numeral, a punctuation symbol, a carriage return, or *Ctrl-C*—is stored as a single byte (8 bits). The values for characters are assigned by the ASCII (American Standards Committee for Information Interchange) code. The first 128 values are pretty much the same throughout the industry, but IBM PC-compatible machines use the second 128 values (128 to 255) for special graphics and line-drawing characters. In C, you use the type **char** (which means the same as **signed char**) to access the values from 0 to 127, with room for negative values for special purposes (such as indicating an error or the end of a file). If you want to access the full character set of the PC, use the type **unsigned char**.

## Input and output for single characters

Here is one way to get a character from the keyboard and store it in a variable (to try it out, load and run INTRO8.C):

```
/* INTRO8.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char inbuf[130];
   char one_char;
   printf("Enter a character: ");
   gets(inbuf);
   sscanf(inbuf, "%c", &one_char);
   printf("The character you entered was %c\n", one_char);
   printf("Its ASCII value is %d\n", one_char);

   return 0;
}
```

A sample run looks like this:

```
Enter a character: A
The character you entered was A
Its ASCII value is 65
```

*Remember that you need the address of one_char, so you must prefix one_char with the **&** (address of) operator.*

*one_char* is a variable of type **char**. The **sscanf** statement uses the specifier **%c** to indicate a single character, and stores the entered value in *one_char*. The next statement displays the value of *one_char* (notice the **%c** conversion specifier for **printf**). The next **printf** statement prints the ASCII code for the entered character. It does this by using the **%d** conversion specifier. This converts the character value to its ASCII code. A character *is* an integer, but certain facilities in C (such as the **%c** specifier) display the value as a character rather than as an integer.

An alternate way to get a character from the keyboard is to use library functions **getch** or **getche**. The previous program could be rewritten as follows:

*This one is INTRO9.C; load and run it to see how it works.*

```
/* INTRO9.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <conio.h>

int main()
{
   int one_char;
   printf("Enter a character: ");
   one_char = getch();
   printf("\nThe character you entered was %c\n", one_char);
   printf("Its ASCII value is %d\n", one_char);

   return 0;
}
```

With this version of the program, the character you type won't be displayed onscreen. If you want the entered character to be visible, use **getche** instead. **gets** reads in an entire line of text before letting the program continue, while **getch** and **getche** read one character, then continue.

## Displaying a character

This example also shows one way to display a character onscreen. The **%c** conversion specifier is used with **printf** to display a single character. Another way to display a single character is with the library function **putch**, which is defined in the header file conio.h. **putch**('>') displays the specified literal character >. If *one_char* is a variable of type **char** (or, in some cases, of type **int**), **putch**(*c*) displays the value of *one_char* as a character.

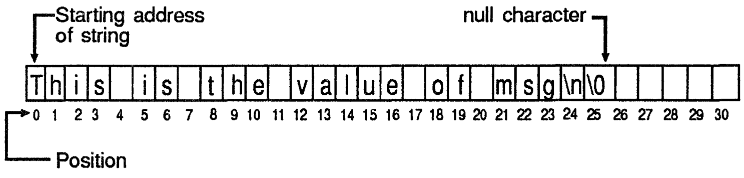
## Displaying character strings

A string is a series of characters. You have already seen strings used in **printf** statements. For example,

```
printf("Enter a character: ");
```

calls **printf** and tells it to display a string of characters beginning with an E and ending with a space. The double quotes tell Turbo C++ to treat the group of characters as a string. Each character is stored in a consecutive byte of memory, and **printf** receives the address of the first character. How does **printf** know when it has reached the end of the string? Whenever you define such a *literal string*, Turbo C++ invisibly tacks a *null character* on the end. This character has an ASCII value of 0, and is represented symbolically as \0. The next figure shows how a string is stored in memory.

Figure 4.2
How a string is stored in memory



```
char message[30];
strcpy(message, "This is the value of msg\n");
```

*Important!* *message* is actually a pointer to a location for the string. Therefore, the code

```
message = "This is the value of msg\n";
```

makes *message* point to where that string is. It does *not* copy the literal string into the storage area of *message*. This is why you must use **strcpy** to copy the characters of the string to the variable.

For convenience, Turbo C++ also provides the library function **puts**. This function writes a string to the standard output, normally the screen. Thus `puts("Enter a character: ")` works almost the same as the **printf** statement just described, with one exception: **puts** automatically puts a new line (**\n**) at the end of the string, advancing the cursor to the next line. If your program doesn't need the advanced formatting capabilities of **printf**, you can save considerable program space by using the simpler **puts** to display strings.

There is an important difference between double quotes and single quotes for specifying strings and characters.

- *"a"* is a *string* consisting of one character, *a*, and the invisible null character
- *'a'* is the single *character a.*

Since strings and characters are different data types, specifiers or functions that work with strings do not work with characters, and vice versa. To display *"a"*, use **printf** with **%s** or **puts**; for *'a'*, use **printf** with **%c** or **putch**.

## Testing conditions and making choices

You have now learned many elements of the C language. So far, all programs have run straight through from beginning to end. Often, however, programs must make choices based on certain values. For example, consider this code fragment:

```
if (bill > credit_limit)
   puts("Consult with the manager");
```

If the amount of the bill is greater than the amount of the credit limit, the code displays the message about consulting with the manager.

### Using relational operators

The **>** (greater than) operator in the preceding statement is a *relational operator*. It expresses a relationship between two values—in this case, whether *bill* is greater than *credit_limit*. Computers use a simple two-valued logic: If a relationship is true, it has a value of 1; if it is false, it has a value of 0. The relational operators are listed in the next table.

Table 4.7
Relational operators

| Operator | Meaning | Example |
|---|---|---|
| **>** | Greater than | 5 > 4 |
| **>=** | Greater than or equal to | 5 >= *x* |
| **<** | Less than | 4 < 5 |
| **<=** | Less than or equal to | *x* <= 5 |
| **==** | Equal to | 5 == 5 |
| **!=** | Not equal to | 5 != 4 |

This simple program shows you how relational operators can be used:

*To try out this program, load and run INTRO10.C.*

```
/* INTRO10.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char inbuf[130];
   int first, second;
   printf("Input two numbers\n");
   gets(inbuf);
   sscanf(inbuf, "%d %d", &first, &second);
   printf("first > second has the value %d\n", first > second);
   printf("first < second has the value %d\n", first < second);
   printf("first == second has the value %d\n", first == second);

   return 0;
}
```

Here's a sample run, using the values 3 and 5 (be sure to type a space between the 3 and the 5):

```
Input two numbers:
first > second has the value 0
first < second has the value 1
first == second has the value 0
```

Notice that a relational test is an expression, since it gives a value. Thus, it can be displayed by **printf**, and you can assign it to a variable with a statement like `hot = (temperature > 90)`. Be careful not to confuse **==** (the relational equals operator) with **=** (the assignment operator). Try editing the last statement in the example so that it reads

```
printf("first == second has the value %d\n", first = second)
```

The expression *first = second* evaluates to the value of *second*, or 5 in the sample run. The **if**, **for**, and other statements that test conditions consider any nonzero condition to be true. You can see that using **=** where you meant **==** will cause inappropriate program behavior (such as being stuck in an endless loop).

## Using logical operators

You can combine more than one condition in a test. To do so, use one of the three *logical operators* shown in the next table.

Table 4.8
Logical operators

| Operator | Meaning |
|---|---|
| && | AND (both conditions must be true) |
| \|\| | OR (at least one condition must be true) |
| ! | NOT (reverse the truth value of a condition) |

For example, the conditional expression

```
(employee_type == temporary) && (wage > 6.00)
```

is true only if the employee is a temporary *and* his or her wage is over $6.00 an hour. C is efficient at handling these operators: If the first condition (`employee_type == temporary`) is found to be false, the second condition isn't tested, since an AND expression is false if either condition is false.

The expression

```
(employee_type == temporary) || (employee_type == hourly)
```

is true if *either* of the two conditions is true. Thus if the first condition is found to be true, there's no need to test the second.

## Branching with if and if...else

Now that you've surveyed relational and logical operators, it's time to put them to work. The simple **if** statement takes the form

> **if** (*conditional expression*)
> *statement* or *group of statements;*

The condition can be a single relational expression or a combination of expressions joined by logical operators. It must be enclosed in parentheses. The **if** statement acts according to the true or false value from the conditional expression. If the expression is true, the statement or group of statements that follow is executed. If you want a group of statements to be executed, enclose them in braces. The following program uses two **if** statements to tell you whether the number you entered is odd or even:

*To try out this code, load and run INTRO11.C.*

```
/* INTRO11.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char  inbuf[130];
   int   your_number;
   printf("Enter a whole number: ");
   gets(inbuf);
```

```
    sscanf(inbuf, "%d", &your_number);

    if (your_number % 2 == 0)
       printf("Your number is even\n");

    if (your_number % 2 != 0) {
       printf("Your number is odd.\n");
       printf("Are you odd, too?\n");
    }
    printf("That's all, folks!\n");

    return 0;
 }
```

After prompting for and storing *your_number*, the program uses an **if** statement to test whether the number is even, using the modulus (**%**) operator. (Since all even numbers are evenly divisible by 2, any even number mod 2 gives a result of 0.) If the number is even, the first **printf** statement is executed. The second **if** statement tests whether *your_number* is odd, that is, if it has a nonzero remainder when it's divided by 2. If the number is odd, you see the following:

```
Your number is odd.
Are you odd, too?
```

Both **printf** statements are executed, since they are grouped together with a set of braces. Finally, the message "That's all, folks!" is displayed. Since it isn't part of an **if** statement, it is executed regardless of whether *your_number* is even or odd.

## Multiple choices with if...else

The previous example probably looked awkward to you. If a number is even, it *can't* be odd, so why make two separate tests? This example can be rewritten much more compactly by adding an **else** branch to the **if**. The **if ... else** statement has this form:

**if** (*conditional expression*)
    *statement* or *group of statements;*
**else**
    *alternative statement* or *group of statements;*

Applying this to the previous example, you get

*To try out this program, load and run INTRO12.C.*

```
/* INTRO12.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
    char inbuf[130];
    int  your_number;
    printf("Enter a whole number: ");
    gets(inbuf);
    sscanf(inbuf, "%d", &your_number);

    if (your_number % 2 == 0)
       printf("Your number is even\n");
    else {
       printf("Your number is odd.\n");
       printf("Are you odd, too?\n");
    }
    printf("That's all, folks!\n");

    return 0;
}
```

You can nest **if** and **else** clauses as deeply as you wish, although after a few dozen nested **if** statements the compiler is likely to run out of memory. Suppose you had to write a program that allocated computers to employees, subject to the following conditions.

- If your employees are programmers and have been working at least two years, give them an 80386 PC.
- If your employees are programmers and have been working *less than* two years, give them an 80286 PC.
- If your employees have been working at least two years, but aren't programmers, give them an 8088 PC.
- Finally, if your employees don't meet any of these conditions, give them a Macintosh.

Try to map out how you could specify these conditions with **if** and **else** statements. Here's one way:

```
if (employee_type == PROGRAMMER) {
   if (years_worked >= 2)
      give_employee(PC386);
   else
      give_employee(PC286);
}

else if (years_worked >= 2)
   give_employee(PC88);
else
   give_employee(Mac);
```

Notice how we used indentation to show which conditions depend on other conditions. First the program determines whether the employee is a programmer. If so, it checks the number of years worked, and awards the appropriate computer. If the employee isn't a programmer, the outer **if...else** statement checks the number of years worked, and awards the machines assigned to nonprogrammers of varying seniority.

## Multiple choice tests: switch

A long series of **if** and **else if** statements is tedious to write, confusing, and prone to error. Consider the next program, which has to decide how to graph a set of data based on the character the user has entered in response to a menu. Here's one way to do it:

*To try out this code, load and run INTRO13.C.*

```
/* INTRO13.C--Example from Chapter 4 of Getting Started */

#include <conio.h>
#include <stdio.h>
#include <ctype.h>

int main()
{
   char cmd;

   printf("Chart desired: Pie  Bar  Scatter  Line  Three-D");
   printf("\nPress first letter of the chart you want: ");
   cmd = toupper(getch());
   printf("\n");

   if (cmd == 'P')
      printf("Doing pie chart\n");
   else if (cmd == 'B')
      printf("Doing bar chart\n");
   else if (cmd == 'S')
      printf("Doing scatter chart\n");
   else if (cmd == 'L')
      printf("Doing line chart\n");
   else if (cmd == 'T')
      printf("Doing 3-D chart\n");
   else printf("Invalid choice.\n");

   return 0;
}
```

The program displays a menu line, then gets a value for *cmd* via the **getch** function. Along the way, the value is passed to the **toupper** function. This ensures that you only need to deal with uppercase characters. The series of **if** and **else if** branches then test for each valid value and execute the corresponding function. The last **else** serves as the *default* case, handling invalid values.

The **switch** statement makes these multipath branches easier to code. It uses the form

*The ... means that you can have as many **case** clauses as you want.*

> **switch**(*value*)
> {
>     **case** *value* : *statement* or *group of statements*
>     ...
>     **default** : *statement* or *group of statements*
> }

The value is tested against the value for the first **case**. If they are the same, the program code given after the colon for the first case is executed until the end of the **switch** statement or until the special statement **break** is reached. If they are different, the value for the next **case** is tested, and so on. If none of the values are the same as the **switch** value, the statement or group of statements following **default** is executed. The **default** is optional. If you don't supply one, and no condition is met, then no statements within the **switch** are executed.

The preceding program example can be rewritten using a **switch** as follows:

*To try out this program, load and run INTRO14.C.*

```
/* INTRO14.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char cmd;

   printf("Chart desired: Pie  Bar  Scatter  Line  Three-D");
   printf("\nPress first letter of the chart you want: ");
   cmd = toupper(getch());
   printf("\n");

   switch (cmd)
   {
      case 'P': printf("Doing pie chart\n"); break;
      case 'B': printf("Doing bar chart\n"); break;
      case 'S': printf("Doing scatter chart\n"); break;
      case 'L': printf("Doing line chart\n"); break;
      case 'T': printf("Doing 3-D chart\n"); break;
      default : printf("Invalid choice.\n");
   }

   return 0;
}
```

The **break** statement at the end of each case is very important. It causes execution to jump past the end of the **switch** statement.

You usually want to include a **break** statement as the last statement for each **case**. For example, remove the **break** at the end of the statement for `case 'L'` and run the program again. If you select *L*, you'll see

```
Doing line chart
Doing 3-D chart
```

The statements for both the *L* and *T* cases are executed. In other words, if you leave out the **break**, execution continues until it finds a **break** or the end of the **switch** statement.

Sometimes this behavior can be useful. Suppose that you want the user of your program to be able to use either *D* (for delete) or *E* (for erase) to remove the current file. You could then code

```
switch (cmd)
{
   case 'I': insert_file(); break;
   case 'F': format_file(current_file); break;
   case 'D':
   case 'E': erase_file(current_file);
}
```

Since there is no **break** statement for `case 'D'`, **erase_file** will be executed for this case as well as for `case 'E'`.

## Repeating execution with loops

The most significant characteristic of the **if**, **else**, and **switch** statements is that they perform their test only once, and execute whatever statements are specified only once. But many computer tasks involve *repetition*; they involve instructions such as "use the same process on each item in this file until you get to the end of the file" or "use the same process on each item in this set of data." For this kind of task, you'll want to use a loop. *Loops* cause a statement or series of statements to be executed repeatedly, monitoring a specified condition in order to determine when to stop. C provides three kinds of loops: **while, do,** and **for**.

### The while loop

The **while** loop executes one or more statements as long as a specified condition is true. The syntax is

**while** (*condition*)

*statement* or *group of statements;*

The following program lets you enter numbers from the keyboard. It keeps a running total. When you enter a 0, it gives you the total and average of the numbers entered.

*To try out this example, load and run INTRO15.C.*

```
/* INTRO15.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char inbuf[130];
   int  number;            /* Number entered by user */
   int  total = 0;         /* Total of numbers entered so far */
   int  count = 1;         /* Count of numbers entered */

   printf("Enter 0 to quit\n");
   /* Priming statement puts a value into number */
   gets(inbuf);
   sscanf(inbuf, "%d", &number);

   while (number != 0)
   {
      total += number;
      gets(inbuf);
      sscanf(inbuf, "%d", &number);
      if(number == 0)
         printf("Thank you. Ending routine.\n");
      else count++;
   }
   printf("Total is %d\n", total);
   printf("Average is %d\n", total / count);

   return 0;
}
```

Once the first number is entered, it is tested by the **while**. As long as the number isn't 0, three things are done:

- The number just entered is added to *total.*
- A new number is obtained with **sscanf**.
- That number is tested. If it isn't 0, the count of numbers is incremented by one.

## The do while loop

The **do while** loop is very similar to the **while** loop. It takes the form

**do** *statement* or *group of statements*

**while** (*condition is true*)

*Important!* What's the difference between a **do while** and a **while** loop?

- The **while** loop performs the test *first* and executes the enclosed statements only if the result of the test is true.
- The **do while** loop executes the enclosed statements and *then* performs the test. This means that the enclosed statements are performed at least once, even if the test turns out to be false.

A good situation for using the **do while** loop is processing a menu. The earlier menu examples had the drawback that they only executed once, and then unceremoniously dumped the user out of the program. Here is the menu program rewritten to use a **do while** statement:

*To try out this example, load and run INTRO16.C.*

```
/* INTRO16.C--Example from Chapter 4 of Getting Started */

#include <conio.h>
#include <ctype.h>
#include <stdio.h>

int main()
{
   char cmd;

   do {
      printf("Chart desired: Pie  Bar  Scatter  Line  Three-D
Exit");
      printf("\nPress first letter of the chart you want: ");
      cmd = toupper(getch());
      printf("\n");

      switch (cmd)
      {
         case 'P': printf("Doing pie chart\n"); break;
         case 'B': printf("Doing bar chart\n"); break;
         case 'S': printf("Doing scatter chart\n"); break;
         case 'L': printf("Doing line chart\n"); break;
         case 'T': printf("Doing 3-D chart\n"); break;
         case 'E': break;
         default : printf("Invalid choice. Try again\n");
      }
   } while (cmd != 'E');

   return 0;
}
```

What has changed? The active part of the program, including the statements that display the menu and get a character as well as the **switch** statement, have been enclosed in a **do while** loop. An additional menu case, *E*, allows the user to exit the program. This **case** simply has a **break** associated with it. If any other character (including an invalid character) had been typed, the **while** condition causes the menu to be redisplayed. But if *E* (or *e*) is typed, the condition `cmd != 'E'` is false, and control drops out of the bottom of the **do while** loop. The program then terminates.

## The for loop

The **for** loop steps through a series of values, performing the specified actions once for each value. The form for this statement is

```
for (starting values; condition; changes)
{
    statement or group of statements;
}
```

The following **for** loop displays the visible characters from the PC's character set (load and run INTRO17.C):

```
/* INTRO17.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int ascii_val;

   for (ascii_val = 32; ascii_val < 256; ascii_val++)
   {
      printf("\t%c", ascii_val);
      if (ascii_val % 9 == 0)
         printf("\n");
   }

   return 0;
}
```

The variable *ascii_val* is the *counter variable* for the **for** loop. The initialization (starting) value is `ascii_val = 32`. The condition `ascii_val < 256` is the *limit* for the loop. The *change* that is made in the counter variable is `ascii_val++` (in other words, it is incremented by one each time through the loop.)

The **printf** and **if** statements are enclosed by braces and make up the body of the loop. The **printf** statement displays the character corresponding to the current value of *ascii_val*, using a tab character (**\t**) for spacing. The **if** statement begins a new line whenever *ascii_val* is evenly divisible by 9—in other words, it ensures that each row will have 9 characters.

There are many variations on the theme of **for** loops. The body of the loop can have only one statement, in which case the braces are optional but recommended for clarity. A **for** loop can have no body at all, with all of the work being done in the change part of the control statement. For example, this loop totals up the numbers from 1 through 10 (load and run INTRO18.C):

```
/* INTRO18.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int number, total;
   for (number = 1, total = 0; number < 11; total += number,
  number++);
   printf("Total of numbers from 1 to 10 is %d\n", total);

   return 0;
}
```

This example also shows that the starting and change parts of the loop specification can have multiple expressions, separated by commas. The loop initializes two variables, *number* and *total*. Each time the loop runs, it adds *number* to *total* and then increments *number*.

*Some programmers put the semicolon on a separate line, so it will stand out.*

The semicolon following the closing parenthesis represents the empty body of the loop. If it is omitted, the loop will grab the **printf** statement and execute it repeatedly, treating it as the body. A **for** loop could be written instead as a **while** loop. The previous example could be rendered as

*To try this out, load and run INTRO19.C.*

```
/* INTRO19.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int number = 1, total = 0;
   while (number < 11) {
      total += number;
      number++;
   }
   printf("Total of numbers from 1 to 10 is %d\n", total);

   return 0;
}
```

The **while** loop performs its initialization before the loop begins, and updates the counter variable within the body of the loop. The **for** loop performs both of these operations within the loop specification itself. The **for** loop is more compact, but it can be harder to read if you cram too many expressions between the parentheses.

## Break and continue

Sometimes it's necessary to bail out of the statements in a loop even before you test the condition again. The **break** statement has two different uses: to break out of a **switch** statement after statements provided for a particular **case** have been executed, and to exit a **while, do while**, or **for** loop immediately, without performing the rest of the statements enclosed in the loop. For example, suppose you didn't want to launch a space shuttle if any warning lights were on:

*To try out this program, load and run INTRO20.C.*

```
/* INTRO20.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

#define WARNING -1

int get_status(void)
{
   return WARNING;
}

int main()
{
   int count = 10;
   while (count-- > 1)
   {
      if (get_status() == WARNING)
         break;
      printf("%d\n", count);
   }
   if (count == 0)
      printf("Shuttle launched\n");
   else
   {
      printf("Warning received\n");
      printf("Count down held at t - %d", count);
   }

   return 0;
}
```

A **while** loop runs the countdown, each time checking the function **get_status** to see if it returns a value of –1 (which you have defined as WARNING). This function would presumably be reading the warning system in real time. If **get_status** returns –1 at any time during the countdown, **break** stops the countdown. Following the **while** statement, the **if** statement checks to see if *count* reached zero. If it did, then no warning occurred. If *count* is greater than 0, however, the countdown must have been interrupted, and the shuttle is not launched.

The **continue** statement, like **break**, causes all remaining statements in the loop to be skipped. But while **break** completely exits the loop, **continue** simply skips to the loop's test condition. INTRO21.C displays the even numbers up through 10:

*To try out this program, load and run INTRO21.C.*

```
/* INTRO21.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   int num = 0;
   while (num++ <= 10)
   {
      if (num % 2 != 0)
         continue;
      printf("%d\n", num);
   }

   return 0;
}
```

If the number is odd, the **continue** statement causes the **printf** statement to be skipped.

**continue** isn't used very often. You have to think the problem through to see when a **continue** or a **break** is appropriate. The use of **continue** inside the **while** loop in the space shuttle program would not be a good idea, since not only would the countdown not be displayed (the **printf** statement in the loop would be skipped), but also *count* would continue down to 0, since the decrement is part of the condition `(count-- > 1)`. The shuttle would be launched even if a warning had been received.

## The goto statement

Veteran BASIC or FORTRAN programmers are familiar with the **goto** statement. C has this statement, too. It has the form

**goto** *label*

where *label* is an identifier that is associated with a particular statement. When **goto** is executed, control jumps to the labeled statement. In early BASIC, **goto** was a necessity because there was no other way to code a loop. With C's three kinds of loops, and the use of **break** and **continue** to skip parts of loops when necessary, there is little need for a **goto** statement. Modern programmers avoid **goto** because it makes a program hard to read and modify. It is too easy to forget why the jump to a particular statement was made. Besides, no language has ever provided a **wherefrom** statement.

## Nested loops

One of the statements in the body of a loop can be another loop—this is called *nesting*. In this example, a **while** loop accepts strings that you enter, and the **for** loop prints hyphens under the string in order to underline it.

*To try out this code, load and run INTRO22.C.*

```
/* INTRO22.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <string.h>
#include <conio.h>

int main()
{
   int pos;
   char text [40];

   printf("Type 'end' to quit\n");

   while (strcmp (gets(text), "end") != 0) {
      for (pos = 1; pos <= strlen(text); pos++)
         putch('-');
      printf("\n");
   }

   return 0;
}
```

This is an example of the powerful but compact style of veteran C programmers. To figure out how the **while** loop is controlled, read from the inside out. **gets**(*text*) gets the input string and stores it in the character array *text*. This function also *returns* the string it has fetched to the calling statement, so that that string can now be compared to the string "end" by the **strcmp** function, which compares two strings. If the comparison yields a 0, the strings are identical, the user had typed `end`, and the loop exits. Otherwise, the body of the **while** loop is executed. The first statement of the body is a **for** loop that prints a number of hyphens equal to the length of the string that was originally entered, using the **strlen** function to find out how long it was. The **printf** statement, which ends the body of the **while** loop, positions the cursor for entry of the next line.

### Choosing appropriate loops

You have now seen three different ways to code loops (**while, do while**, and **for**). Use whichever form of statement expresses the idea of the program most clearly, keeping in mind the following guidelines:

- If you don't want the body of the loop to be executed at all if the condition is false, use a **while** loop.
- If you want the body of the loop to always be executed at least once, use a **do while** loop.
- If the number of times the loop is to be executed is determined by the value of a variable or constant, it is usually best to use a **for** loop.
- If the loop is to be executed as long as some externally determined condition is true (for example, as long as there is data left in the file), use a **while** loop.

## Program design with functions and macros

Now that you know how to control the execution of a program, you can do some useful work in C. We encourage you to modify and elaborate upon the example programs. Small, simple programs such as the ones we've covered so far don't need a lot of structure. However, as your programs grow larger and more complex, you need to break them down into smaller, more manageable logical pieces, or functions.

### Defining your own functions

The programs you have already seen perform divisions of labor. When you call **gets, puts,** or **strcmp**, you don't have to worry about how the innards of these functions work. These and about 400 other functions are already defined and compiled for you in the Turbo C++ library. To use them, you need only include the appropriate header file in your program and check the online help or Chapter 1, "The run-time library," in the *Library Reference* to make sure you understand how to call the function, and what value (if any) it returns.

But you'll need to write your own functions. To do so, you need to break your code into discrete sections (functions) that each perform a single, understandable task for your program. Once you have declared and defined your own functions, you can call them throughout your program in the same way that you call Turbo C++ library functions.

## The function prototype

Function prototypes are a key feature of the new ANSI standard for the C language. A function *prototype* is a declaration that takes this form:

*return_type function_name (parameter_type parameter_name ...);*

Here are some examples:

*Note that* ***main****, as a function itself, must also conform to the ANSI recommendation that all functions be given prototypes.*

```
int main(void)
float tax(float purchase);
char get_employee_type(int employee_num);
char get_choice(void);
int getch(void);
void show_menu(void);
```

By looking at the prototype, you can tell exactly what type of information the function expects, and what type it returns.

Let's look at **tax**. When you call it, you need to give it a floating-point number. The result will also be floating point. The prototype informs Turbo C++ about all this.

Some of the other example prototypes shown earlier use the keyword **void**. **void** means empty, or none. When the word **void** appears in the parentheses in place of the parameter list, this indicates that the function has no parameters. Thus if you tried to use the statement `show_menu(10)`, you would be informed that **show_menu** doesn't take any parameters. When **void** appears as the function's return type, it means that the function does not return a value—so you shouldn't try to assign the function call's result to a variable.

### Function declarations under Kernighan and Ritchie

*We recommend that you use prototypes in your code.*

In the old Kernighan and Ritchie style, the return type of a function was given only if it wasn't **int**. Similarly, function parameters were declared within the body of the function definition, rather than in the parameter list. The **get_employee_type** function would be declared in this old style as

```
char get_employee_type();
```

And the function's actual code (its definition) would look like this

```
char get_employee_type(employee_num)
int employee_num;
{
   /* body of function code */
}
```

Old-style functions compile correctly under Turbo C++. However, they have less information about the function's parameters, so errors involving the wrong type of parameters in function calls won't be caught automatically. This is one of the many good reasons why this older style is being discarded by ANSI.

## The function definition

The function definition contains the actual executable code for the function. The preferred form for the function header starts out exactly the same as the function declaration, except that it doesn't end in a semicolon. It is followed by local variable declarations and the code to be executed, enclosed in braces.

```
float tax(float purchase)
{
   float tax_rate = 0.065;
   return(purchase * tax_rate);
}
```

## Processing within the function

The function sees its parameters as though they had been declared as variables of the indicated types. The **tax** function thus has access to two values: its **float** parameter, *purchase*, and its own **float** variable, *tax_rate*. If you make the function call `tax(amount)`, it is a *copy* of the value of *amount* that the function **tax** receives through its parameter *purchase*. The function refers to this value under the name *purchase* and can change the value of *purchase*. This doesn't change the value of the original variable *amount*, however. We'll show you later how a function can use pointers to change the values of variables used to call it.

## The function return value

A function doesn't have to return a value—in that case, you should declare its return type to be **void**. To return a value to the caller, a function uses the **return** statement (as in the function **tax**, where the value returned is *purchase * tax_rate*). The next figure summarizes how information flows to the **tax** function and then flows back again.

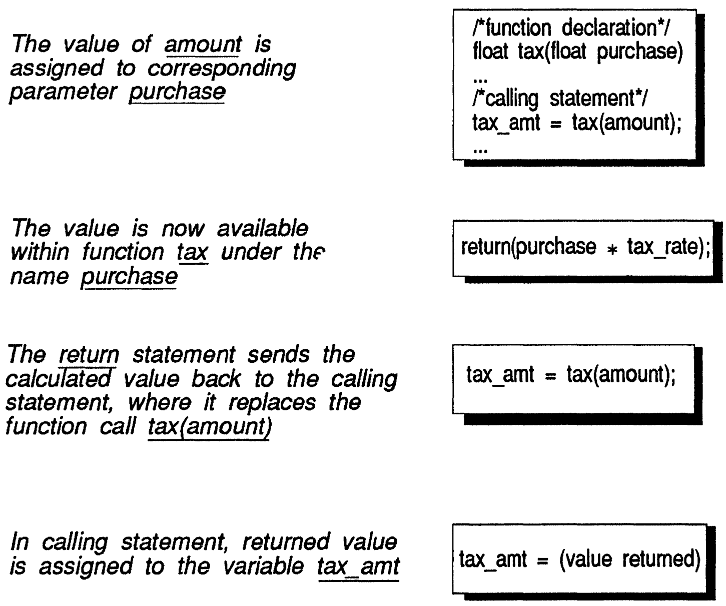


Figure 4.3
Information flow to and from the tax function

*The value of amount is assigned to corresponding parameter purchase*

```
/*function declaration*/
float tax(float purchase)
...
/*calling statement*/
tax_amt = tax(amount);
...
```

*The value is now available within function tax under the name purchase*

```
return(purchase * tax_rate);
```

*The return statement sends the calculated value back to the calling statement, where it replaces the function call tax(amount)*

```
tax_amt = tax(amount);
```

*In calling statement, returned value is assigned to the variable tax_amt*

```
tax_amt = (value returned)
```

## Using the return value

The returned value of a function can be treated like any other value in an expression. It can be combined with other variables and arithmetic operators in an expression; it can be part of the condition for an **if** statement or loop; it can be assigned to a variable, and so on. Here are some examples of the use of function return values that you have already seen:

```
tax_amt = tax(purchase);
if (get_status() == WARNING)
while (strcmp(gets(text), "end") != 0)
```

In the first example, the value returned by the **tax** function is assigned to the variable *tax_amt*. In the second example, the value returned by the call to **get_status** is compared with the defined value WARNING; the result determines the true value of the **if** statement. In the last example, the value returned by **gets** is passed to the **strcmp** function (along with the string "end"), and the result of the call to **strcmp** in turn is compared with zero. That result becomes the value of the **while** statement test.

## Multifunction programs

The following program draws a graphic representation of part of the solar system. It illustrates the use of several user-defined functions as well some features of Turbo C++'s graphics library. (For more on how to use Turbo C++ graphics, see Chapter 5, "Video functions," in the *Programmer's Guide.*)

*Be sure to set your path argument in calls to* ***initgraph*** *to the directory where your .BGI files are.*

As written, the program requires EGA or VGA graphics hardware, but because it scales itself to the capabilities of the adapter, you could run a cruder version in CGA if you change the color constants used. The output is shown in the next figure.

The program listing has function prototypes, global declarations, the definition for **main**, and then the definitions of the various other functions. This program is included on your disk as PLANETS.C.

*After reading through the discussion of this program, feel free to modify it. Study the graphics library. Try different colors and fill styles. Experiment with various scaling values for distances and radii.*

```
/* PLANETS.C--Example from chapters 4 and 7 of Getting Started */

#include <graphics.h>          /* For graphics library functions */
#include <stdlib.h>            /* For exit() */
#include <stdio.h>
#include <conio.h>

int set_graph(void);           /* Initialize graphics */
void calc_coords(void);        /* Scale distances onscreen */
void draw_planets(void);       /* Draw and fill planet circles */

/* Draw one planet circle */
void draw_planet(float x_pos, float radius,
                 int color, int fill_style);
void get_key(void);        /* Display text on graphics screen, */
                           /* wait for key */

/* Global variables -- set by calc_coords() */
int max_x, max_y;          /* Maximum x- and y-coordinates */
int y_org;                 /* Y-coordinate for all drawings */
int au1;                   /* One astronomical unit in pixels
                              (inner planets) */
int au2;                   /* One astronomical unit in pixels
                              (outer planets) */
int erad;                  /* One earth radius in pixels */

int main()
```

```
{
   /* Exit if not EGA or VGA */
   /* Find out if they have what it takes */
   if (set_graph() != 1) {
      printf("This program requires EGA or VGA graphics\n");
      exit(0);
   }
   calc_coords();       /* Scale to graphics resolution in use */
   draw_planets();      /* Sun through Uranus (no room for others) */
   get_key();           /* Display message and wait for key press */
   closegraph();        /* Close graphics system */

   return 0;
}

int set_graph(void)
{
   int graphdriver = DETECT, graphmode, error_code;

   /* Initialize graphics system; must be EGA or VGA */
   initgraph(&graphdriver, &graphmode, "..\\bgi");
   error_code = graphresult();
   if (error_code != grOk)
      return(-1);               /* No graphics hardware found */
   if ((graphdriver != EGA) && (graphdriver != VGA))
   {
      closegraph();
      return 0;
   }
   return(1);                   /* Graphics OK, so return "true" */
}

void calc_coords(void)
{
   /* Set global variables for drawing */
   max_x = getmaxx();           /* Returns maximum x-coordinate */
   max_y = getmaxy();           /* Returns maximum y-coordinate */
   y_org = max_y / 2;           /* Set Y coord for all objects */
   erad = max_x  / 200;         /* One earth radius in pixels */
   au1 = erad * 20;             /* Scale for inner planets */
   au2 = erad * 10;             /* scale for outer planets */
}

void draw_planets()
{
   /* Each call specifies x-coordinate in au, radius, and color */
   /* arc of Sun */
   draw_planet(-90, 100, EGA_YELLOW, EMPTY_FILL);
   /* Mercury */
   draw_planet(0.4 * au1, 0.4 * erad, EGA_BROWN, LTBKSLASH_FILL);
   /* Venus */
   draw_planet(0.7 * au1, 1.0 * erad, EGA_WHITE, SOLID_FILL);
   /* Earth */
   draw_planet(1.0 * au1, 1.0 * erad, EGA_LIGHTBLUE, SOLID_FILL);
   /* Mars */
   draw_planet(1.5 * au1, 0.4 * erad, EGA_LIGHTRED, CLOSE_DOT_FILL);
   /* Jupiter */
   draw_planet(5.2 * au2, 11.2 * erad, EGA_WHITE, LINE_FILL);
   /* Saturn */
   draw_planet(9.5 * au2, 9.4 * erad, EGA_LIGHTGREEN, LINE_FILL);
   /* Uranus */
   draw_planet(19.2 * au2, 4.2 * erad, EGA_GREEN, LINE_FILL);
}

void draw_planet(float x_pos, float radius, int color, int
fill_style)
{
   setcolor (color);              /* This becomes drawing color */
   circle(x_pos, y_org, radius);  /* Draw the circle */
   setfillstyle(fill_style, color); /* Set pattern to fill interior
*/
   floodfill(x_pos, y_org, color);  /* Fill the circle */
}

void get_key(void)
{
   outtextxy(50, max_y - 20, "Press any key to exit");
   getch();
}
```

## Function prototypes and global declarations

This program calls five programmer-defined functions. Their declarations appear right after the **#include** statements. These declarations could appear elsewhere, but then there wouldn't be any type checking until the prototypes are reached. It is easiest to have them right at the start. The prototypes of **void** and non-**int** functions *must* appear before the first call to those functions.

The global variables hold information needed for drawing the planets. Their values are calculated by the **calc_coords** function, and these values are used by **draw_planets**. Making these variables global (rather than declaring them inside a function) makes them accessible to both functions that need them. Later, we'll show an alternate way to share variables between two functions.

## Setting up the graphics display

**main**'s first call is to **set_graph**, which "packages" a number of operations involving the Turbo C++ graphics library. **set_graph** uses the library function **initgraph** with the DETECT mode to automatically determine what kind of graphics hardware is present. Notice the multiple **return** statements—a function can have as many **return** statements as needed. The first **if** determines whether there was an error initializing the graphics system (the code returned isn't equal to *grOk* ("graphics OK")—the function exits and returns an error code in that case. The second test returns an error code if *neither* EGA nor VGA capability is present. The second test is made only if the first was successful. The use of multiple **return** statements avoids the need for **else** statements.

The identifiers DETECT and *grOk* appear not to have been declared anywhere. In fact, these are defined constants that are part of the graphics.h header file. In addition to function prototypes, header files frequently make useful definitions available to your programs. We recommend that you browse through graphics.h and other header files that you frequently use in your programs, so that you become familiar with their contents. All of the colors and pattern fill styles you'll encounter later are also defined in graphics.h. Each identifier has an integer value associated with it, but the use of symbolic names makes it much easier to see what is going on. (BLACK is much more meaningful than 0.)

**main** checks the value returned by **set_graph**. If anything other than 1 was returned, the program displays a message and exits.

## Calculating the graphics coordinates

If everything checks out, **calc_coords** is called next. Many beginning programmers are used to thinking of the *x-y* dimensions of the graphics screen as being fixed by those provided by the graphics adapter they use—for example, 640x350 for EGA. The temptation is to hard-code these exact dimensions into your program. But this leads to trouble when the program is run on a machine that has a different graphics resolution and set of available colors. As discussed further in Chapter 5, "Video functions" of the *Programmer's Guide*, the graphics library helps you write programs that run on a wide range of graphics adapters.

To help make this possible, the library functions **getmaxx** and **getmaxy** functions return the highest x- and y-coordinates, respectively, for the graphics mode currently set. (This in turn was set by **initgraph**.) The remaining statements

- set the center for the planetary circles by taking half of the maximum *y* value.
- scale the Earth's planetary radius (which is the unit used to express the radii of the other planets) to 1/200th of the width of the screen (the maximum *x* value).
- set two distance measurements for placing the centers of the circles on the x-axis. Because distances in the solar system increase rapidly once past Mars, different scales are used for the inner and outer planets. As a result, the drawing won't be accurate in distances, though it will accurately show the relative sizes of the planets. (Due to the limited size of the screen, you can't have both.) For the same reason, Neptune and Pluto had to be omitted.

## Drawing the planets

The function **draw_planets** gathers together a series of calls to the function **draw_planet**, which does the actual work. **draw_planet** takes four parameters: *x_pos, radius, color,* and *fill_style.*

- *x_pos* is the x-coordinate for the center of the planetary circle. It is obtained by multiplying the distance unit (*au1* or *au2*) by the mean distance of the planet's orbit from the sun, expressed in astronomical units. (An astronomical unit is the distance of the Earth from the Sun, approximately 93 million miles.)
- *radius* is the radius for the planetary circle. This is obtained by multiplying the actual planet's radius in terms of Earth's radius (about 4,000 miles) by *erad*, the number of pixels per Earth radius.
- *color* is a constant from graphics.h that gives the color to be used for drawing the circle (and for filling it in) from the default EGA palette (which also works for VGA).
- *fill_style* is a constant from graphics.h that gives the style to be used for filling in the circle.

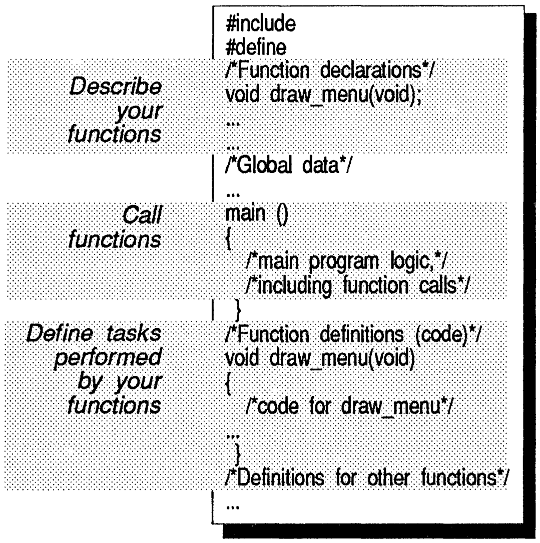
**draw_planet** takes these parameters and calls Turbo C++ graphics library routines to draw the circle and fill it in. Notice that the y-coordinate needed by **circle** and **floodfill** didn't have to be supplied as a parameter. Since it is fixed, the quantity *y_org* previously calculated by **calc_coords** is used.

Finally, **get_key** uses the **outtext** library function to display a message, then calls **getch**, which waits for a key to be pressed to exit the program.

## Header files, functions, and libraries

In a small program, you will probably declare and define your functions in the same file, together with your **main** function that ties everything together. Such a structure is shown in the next figure.

Figure 4.4
Simple program structure (all in one)

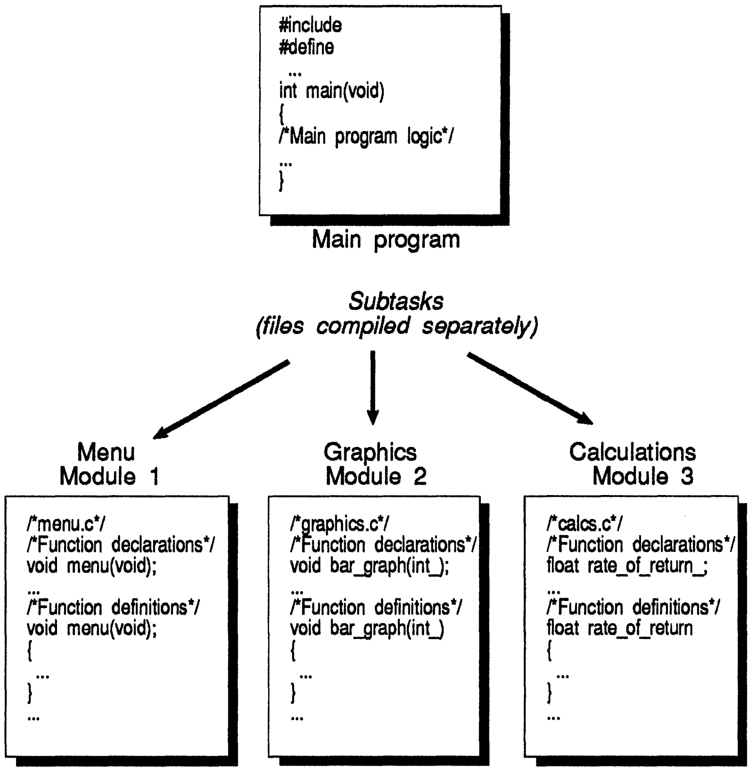


```
#include
#define
/*Function declarations*/
void draw_menu(void);
...
...
/*Global data*/
...
main ()
{
   /*main program logic,*/
   /*including function calls*/
}
/*Function definitions (code)*/
void draw_menu(void)
{
   /*code for draw_menu*/
...
}
/*Definitions for other functions*/
...
```

*Header files usually only contain function declarations so they can be included in many different source or module files.*

As programs get larger, however, it becomes desirable to group related function definitions in a separate file. For example, the functions dealing with the user interface might go into one file, the functions dealing with data processing in another, and the functions dealing with presentation graphics in yet a third file. Turbo C++ can compile all three files together to create the final executable program. This kind of structure is shown in the next figure.

Figure 4.5
Program built from several files



```
#include
#define
...
int main(void)
{
/*Main program logic*/
...
}
```

```
/*menu.c*/
/*Function declarations*/
void menu(void);
...
/*Function definitions*/
void menu(void);
{
...
}
...
```

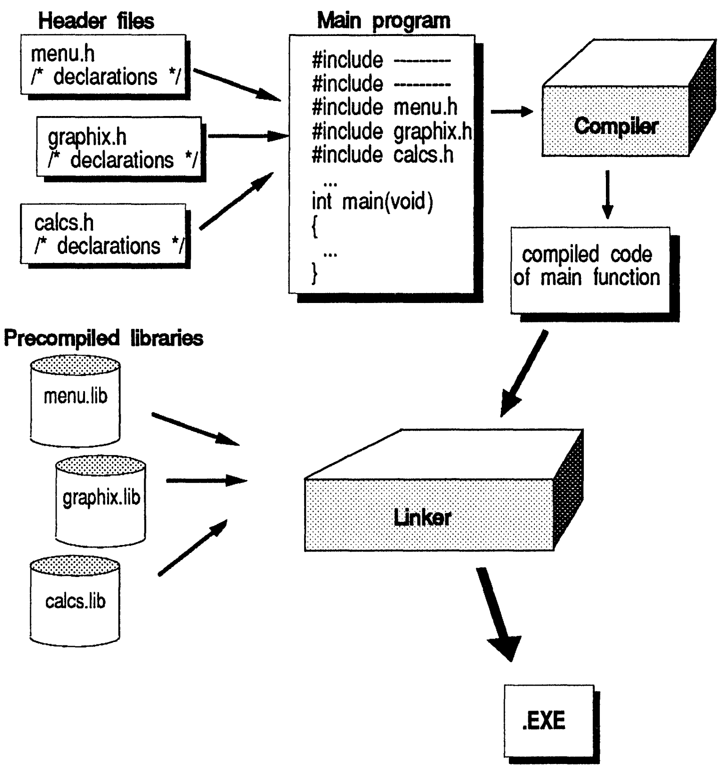
```
/*graphics.c*/
/*Function declarations*/
void bar_graph(int_);
...
/*Function definitions*/
void bar_graph(int_)
{
...
}
...
```

```
/*calcs.c*/
/*Function declarations*/
float rate_of_return_;
...
/*Function definitions*/
float rate_of_return
{
...
}
...
```

*You could substitute an `#include "header.h"` command for the function declarations.*

Once parts of your program are stable, you can compile groups of functions into libraries. The declarations for the functions in each library can be put into a header file like those you use to access Turbo C++'s own libraries. Your main program includes the header files, thus inserting the function declarations into your program text. After compilation, the linker links the libraries into your program's object code. This process is shown in the next figure.

Figure 4.6
Program using custom libraries



## Scope and duration of variables

As programs get more complicated, the question of access to variables used in other parts of the program arises. Every variable has two characteristics: scope and duration. *Scope* (sometimes called *visibility*) defines what parts of a program can access the variable. *Duration* specifies how long the variable remains accessible.

### Scope

Scope is determined by *where* you declare the variable. A variable defined within a function definition is by default *local*—it can only be accessed by code within the same function. For example, in this program, *You can load and run this program: INTRO23.C.*

```
/* INTRO23.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

void showval(void);

int main()
{
   int mainvar = 100;
   showval();
   printf("%d\n", funcvar);

   return 0;
}

void showval(void)
{
   int funcvar = 10;
   printf("%d\n", funcvar);
   printf("%d\n", mainvar);
}
```

the function **showval** first uses a **printf** statement to display the value of *funcvar*. This is fine, since *funcvar* is declared and defined within the **showval** function. The next statement, which attempts to display the value of *mainvar*, causes the compiler to complain (by way of an error message) that *mainvar* is *undefined*. This is because *mainvar* is defined *within* another function, namely **main**. It cannot be accessed from within **showval**.

Even if you fix this, upon return from the call to **showval**, the **main** function tries to display the value of the variable *funcvar*. This, too, will cause an error because *funcvar* is defined within **showval**.

To make a variable visible from within *any* function in the current source file, define it *outside* any function definition. It will be visible after the position in the source file where it is declared, so the usual place to define such *global* variables is before the start of the definition of **main**. If the preceding example starts out like this,

```
void showval(void);
int mainvar, funcvar;
```

and if you remove the **int** declaration from

`int mainvar = 100` and
`int funcvar = 10,`

there will be no complaint. Remember, however, that a variable that is accessible from anywhere is also changeable from anywhere, which can lead to bugs that are hard to track down. Changes in value caused this way are sometimes called *side effects*.

By default, global variables are accessible from any file. (Although, without **extern** references in other files, a global variable has file scope.) If you have a program that uses more than one source file, and you need to make a variable visible in a different source file, declare it in the current file by adding the keyword **extern** ("external"). Thus, if the file main.c defines `int xscale`, you can "see" this variable from within another file (such as stars.c) by declaring

```
extern int xscale;
```

there. The variable is assigned its memory address when it is originally defined. The **extern** declarations merely inform the compiler that an external variable will be referenced.

## Duration

It would be inefficient to reserve memory permanently for all of the variables in a large program. After all, a particular function may only be called once. By default, variables declared within a function definition are **auto** (automatic) variables. Their memory is allocated when their function begins to execute. When the function returns to its caller, the memory is freed up for use by other variables.

Occasionally you may wish to override this default behavior and have a variable stored permanently, even when the function it is declared in isn't running. The keyword **static** accomplishes this. For example, a function could count how many times it was called:

*To try out this code, load and run INTRO24.C.*

```
/* INTRO24.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <conio.h>

void tally(void);

int main()
{
   while ( getch() != 'q')
      tally();

   return 0;
}
```

```
void tally(void)
{
   static int called = 0;
   called++;
   printf("Function tally called %d times\n", called);
}
```

Each time the **while** loop in **main** receives a character other than *q* from **getch**, it calls **tally**. **tally** increments the static variable *called* on each call. C initializes static numeric variables to 0 when they are declared.

Another declaration keyword that is occasionally used is **register**. When it is added to a variable declaration, **register** asks the compiler to generate code that uses one of the microprocessor's fast internal registers to hold the value, rather than using the slower random access memory. **register** can only profitably be used with data types small enough to fit in a register, such as **char** or **int**.

Since modern compilers such as Turbo C++ optimize so that they take advantage of machine resources efficiently, only experienced programmers know when to take advantage of this feature. Using it inappropriately can slow down your program. It is usually best to let the compiler figure out whether to use a machine register for a variable. Also, using the keyword **register** doesn't guarantee that the variable will be saved in a register. It is only a suggestion to the compiler that it attempt to do so.

## Using constant values

A *constant* is a value that is fixed—it doesn't change during the execution of your program. There are two ways to define constants: the **const** keyword and the #**define** directive.

1. One way to create constants is to use the keyword **const** in a declaration. For example, if you declare

   ```
   const float cm_per_inch = 2.54
   ```

   you are telling the compiler that this value will never change. If you attempt to assign a new value to it anywhere in your program (including by incrementing or decrementing it with **++** or **- -**), you will receive an error message. Using this keyword therefore helps Turbo C++ to catch programming slips.
2. Another way to include constant values in a program is by using the #**define** directive. You have already seen a few examples, such as

```
#define RATE 0.065
```

This is not a declaration, but an instruction to the preprocessor, a part of Turbo C++ that will make requested changes to your source code before it is compiled. Here, the change is equivalent to using a word processor to find all instances of RATE and replace them with the characters 0.065.

There are times when either **const** or **#define** is more appropriate. In both cases, you can change the value by simply changing the definition and recompiling. **const** gives the important advantage that Turbo C++ knows what data type the value should have (for example, **const int**). On the other hand, a **#define** can appear in multiple modules without problems, but **const float** can't.

The advantage of using constants is that when the value changes, you need change only this one statement. You don't have to search for every instance of a particular number—a process which, besides being tedious, is prone to error.

## Using macros to hide details

**#define** can do more than simplify substitution. You can design macros that take parameters (much in the same way as functions) and embed them in a template of text. For example,

*You can load and run this program: INTRO25.C.*

*Programmers usually use the **? :** expression for macros.*

```
/* INTRO25.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

#define EVEN(value) (((value) % 2 == 0) ? (value) : ((value) + 1))

int main()
{
   char inbuf[130];
   int  num;
   printf("Enter a number: ");
   gets(inbuf);
   sscanf(inbuf, "%d", &num);
   printf("\n%d", EVEN(num));

   return 0;
}
```

*The extra parentheses around value in the macro definition allow for expressions.*

In the statement `printf("\n%d", EVEN(num))`, the definition causes the expression EVEN(*num*) to be replaced in the program text by the following: `((num) % 2 == 0) ? (num) : ((num) + 1)`. In other words, whatever term is used in parentheses replaces the name *value* each place that it occurs within the template. The resulting expression evaluates to *num* if it is evenly divisible by 2 `((num) % 2 == 0)`; otherwise, it is odd and is replaced by `(num) + 1`, making it even.

The EVEN macro could be written as a function, of course:

```
int even (int num)
{
   return (num % 2 == 0 ? (num) : (num + 1));
}
```

*The tradeoff between functions and macros is speed versus space. Functions are usually smaller but slower than macros.*

While the macro and the function are called in the same way, the way they work is actually quite different. For a macro, the template with the inserted value replaces the call to the macro in the actual source code before compilation. With a function, the code for the function is compiled, and the function call is compiled into code that passes the appropriate value(s) on the stack and then jumps to the function's code. The code for the function itself is compiled only once, no matter how many times the function is called. With the macro, new source code is inserted into the file each time the macro is called.

*If you use a macro, make sure the value being substituted is of the appropriate type.*

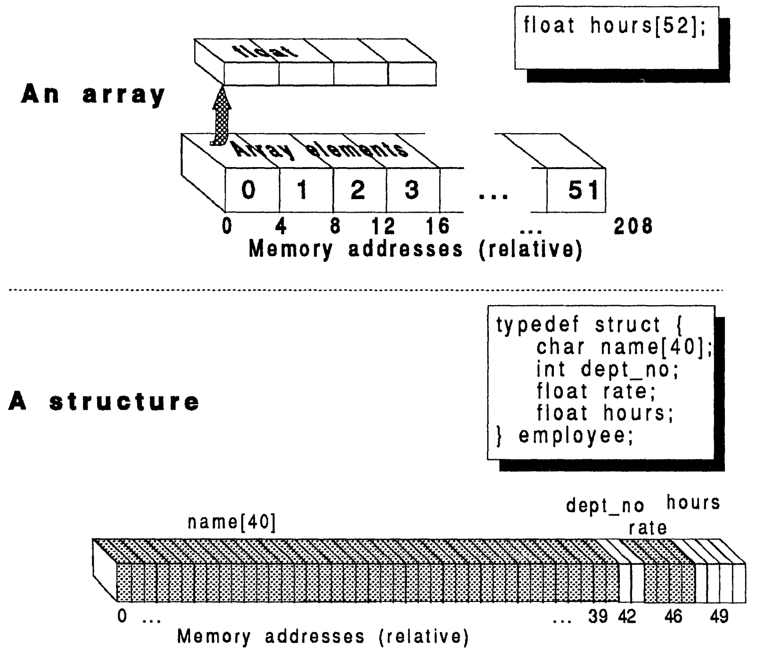
Macros make the source code more readable by replacing a complicated expression with an easy to recognize name. A number of "functions" in the run-time library are really macros; for example, the character classification routine **isalpha** which checks to see whether a character is part of the alphabet as opposed to being a digit, punctuation mark, and so on. You can study this and a number of other macro definitions by looking at the header file ctype.h.

## Building data structures

Data tends to come in bunches rather than single pieces. For example, you may need to keep track of the number of hours an employee has worked each week during the current year. Here you have a set of related data items of the same type (total hours, probably a **float** to allow for fractional hours). C allows you to declare an *array* to store such a set of homogeneous data. But your business also has to keep track of a variety of information about each employee, such as name, years worked, salary, department, and so on. These items are certainly related (they all refer to an employee), but they are of different types. Names are character strings (arrays of characters), years worked can be an **int** or a **float** depending on whether fractions are allowed, the salary is probably a **double** (to allow for well-paid employees), and the department can be a numeric code or a character string.

Figure 4.7
Two ways to deal with sets of data

*Note that the different members of a structure may not always be contiguous in memory.*



```
float hours[52];
```

```
typedef struct {
    char name[40];
    int dept_no;
    float rate;
    float hours;
} employee;
```

In addition to organizing your data, you need a way that you can easily find the particular item you want to work with. Since all this data is actually stored in blocks of memory addresses (something like house addresses on a street), you can point to the data you want by using addresses. In this section, you'll learn how to use pointers to access data structures.

## Declaring and initializing an array

An array is a chunk of memory that is used to hold a group of data items of the same type. For example, an array of **int** will hold the specified number of integers in consecutive memory locations. You specify an array by giving the type of data to be stored, the array name, and the number of items to be stored; put brackets around the number of items to be stored.

**type** *name*[*size*];

Here is how you might declare an array that will hold the total hours worked per week for an employee for one year:

```
float hours[52];
```

This can be read as "hours, an array of 52 **float** values."

*Arrays start at index 0 and end at a position one less than their assigned size.*

A particular item, called an *element*, of the array can be referred to by giving the array name followed by the position of the item, in brackets. The first item is stored at the address pointed to by the array name itself. This can also be referred to as position 0. Thus the total for the first week in the *hours* array can be referred to as *hours*[0]. The total for the tenth week would be *hours*[9], and the total for the 52nd week would be *hours*[51].

The following program initializes the array *hours* to 0, assigns values for the first four positions, and then prints the value out, showing how they are accessed:

*To try this one out, load and run INTRO26.C.*

```
/* INTRO26.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   float hours[52];
   int week;

   /* Initialize the array */
   for (week = 0; week < 52; week++)
      hours[week] = 0;

   /* Store four values in array */
   hours[0] = 32.5;
   hours[1] = 44.0;
   hours[2] = 40.5;
   hours[3] = 38.0;

   /* Retrieve values and show their addresses */
   printf("Elements\t\tValue\tAddress\n");
   for (week = 0; week < 4; week++)
      printf("hours[%d]\t\t%3.1f\t%p\n", week, hours[week],
             &hours[week]);

   return 0;
}
```

The output will look like this:

*The addresses will vary from one time to the next, and from one machine to another.*

```
Element         Value   Address
hours[0]        32.5    FF0E
hours[1]        44.0    FF12
hours[2]        40.5    FF16
hours[3]        38.0    FF1A
```

Notice that the elements are stored in consecutive addresses, 4 bytes apart (which is, after all, the size of a single **float**). The address operator **&** retrieves the address of the element referenced. The **printf** specifier **%p** (for pointer) displays the address as a hexadecimal number. The **for** statement is quite convenient for stepping through the elements of an array.

You can also explicitly initialize an array at the time it is declared. To do so, place the values you want to assign between braces, separating them with commas. For example,

```
int quarters[4] = {3, 10, 7, 14};
```

might hold the amount of points scored by a team in each quarter of a football game. (Of course data like this would probably come from being entered at the keyboard or being read from a file. But you can use explicit assignments to an array to provide data for testing your program.)

When you assign character constants, put each character in single quotes:

```
char grades[5] = { 'A', 'B', 'C', 'D', 'F'};
```

If you specify fewer values than the size of the array will accommodate, and the array is global or static, Turbo C++ sets the remaining elements to 0 (if the array is of the numeric type) or to null characters (in the case of an array of characters). If you specify *more* values than specified in the size of the array, you will receive the error message, "Too many initializers."

Arrays can be more complex than this (for example, you can omit the array size, or initialize a character array with a string). However, such topics are beyond the scope of this chapter.

## Arrays with multiple dimensions

Sometimes it is useful to have a set of sets of values—for example, if you want to store the total hours worked by 12 employees during 52 weeks, you can declare

```
float hours[12][52];
```

Read this as "an array containing 12 arrays of 52 values each, of type **float**." You can think of this layout as being like a spreadsheet with 12 columns and 52 rows.

This next game generates a fictitious baseball score, by using an array called *scoreboard*[2][9] representing two teams and their respective scores for nine innings.

*Load and run GAME.C.*

```
/* GAME.C--Example from Chapter 4 of Getting Started */

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

#define DODGERS 0
#define GIANTS 1

void main(void)
{
   int scoreboard [2][9];    /* An array two rows by nine columns */
   int team, inning;
   int score, total;

   randomize();              /* Initialize random number generator */

   /* Generate the scores */
   for (team = DODGERS; team <= GIANTS; team++) {
      for (inning = 0; inning < 9; inning++) {
         score = random(3);
         if (score == 2)     /* 1/3 chance to score at least a run */
            score = random(3) + 1;   /* 1 to 3 runs */
         if (score == 3)
            score = random(7) + 1;   /* Simulates chance of a big
                                        inning of 1 to 7 runs */
         scoreboard[team][inning] = score;
      }
   }

   /* Print the scores */
   printf("\nInning\t1   2   3   4   5   6   7   8   9   Total\n");
   printf("Dodgers\t");
   total = 0;
   for (inning = 0; inning <= 8; inning++) {
      score = scoreboard[DODGERS][inning];
      total += score;
      printf("%d   ", score);
   }
   printf("   %d", total);

   printf("\nGiants\t");
   total = 0;
   for (inning = 0; inning < 9; inning++) {
         score = scoreboard[GIANTS][inning];
         total += score;
         printf("%d   ", score);
      }
      printf("   %d\n", total);
   }
```

Not surprisingly, when two array dimensions are involved, two nested **for** loops are often used to access the array elements. The inner loop, using *inning* as the counter variable, steps through the nine innings, while the outer switches from team 0 (Dodgers) to team 1 (Giants).

Within the body of the loop, the **random** function (defined in the header file stdlib.h) generates the score. When **random** is first called as `random(3)`, there is a 1/3 chance of *score* getting the value 2. (**random** returns a value between 0 and one less than its parameter). If *score* has the value 2, it is recalculated as a random number between 1 and 3 runs. Finally, if *score* now is 3, a final random score of 1 to 7 runs is generated. The series of **if** statements thus attempts to simulate the occasional big inning.

Two **for** loops then print out the scores, totaling them as they go. Each one prints out one team's scores by using the team name as a constant value for the first dimension of the array, and varying the *inning*. Here's a sample run:

```
Inning  1   2   3   4   5   6   7   8   9   Total
Dodgers 0   1   0   1   0   1   1   2   0     6
Giants  1   1   0   2   0   0   4   1   0     9
```

## Arrays and strings

Strings and arrays are very similar. In fact, a *string* is simply an array of **char** values with a null character stuck on the end. The following program declares a character array (string), then lets you store a value in it and extract a "substring" from the full string (load and run INTRO27.C):

*Remember that string arrays need an extra element for the ending null character.*

```
/* INTRO27.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char string[80];              /* Has 79 usable elements */
   char number[10];
   int pos, num_chars;
```

```
      printf("Enter a string for the character array: ");
      gets(string);
      printf("How many characters do you want to extract? ");
      gets(number);
      sscanf(number, "%d", &num_chars);

      for (pos = 0; pos < num_chars; pos++)
         printf("%c", string[pos]);
      printf("\n");

      return 0;
   }
```

Here's a sample run:

```
Enter a value for the character array: The quick brown fox
How many characters do you want to extract? 9
The quick
```

It is usually more convenient to use the library routines that manipulate strings (such as **strcat** or **strtok**) to deal with strings because they automatically take care of putting a null character at the end of the string. If you create a string with an array declaration and want to use these routines with it, you must supply the null character yourself. (The array must be large enough to hold the desired string *plus* the null character, which you must put at the end of the string.)

## Defining string variables

You can define a variable of type **char** to hold a single character. Can you define a variable of type **string** to hold a string? No, because C does not treat strings as a separate data type. Remember that a string is defined as a series of characters. One way to define a string variable is to declare an *array of characters*. Here's an example of how this works.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char message[30];
   strcpy(message, "This is the value of msg\n");
   puts(message);
   return 0;
}
```

The line `char message[30]` declares an array of characters (a string) that can hold up to 30 characters. You can only use 29 characters, though, because there must also be room for the null character at the end. This chunk of 30 bytes has a starting address, which is stored in *message*. In the **strcpy** call, the literal string

```
This is the value of msg\n
```

is first compiled and stored, and a null character is added at the end.

When **strcpy** runs, it copies the characters from this string, one at a time, into memory, starting at *message*'s address.

The Turbo C++ library has a variety of functions that do useful things with strings—you can read about them in Chapter 2, "Run-time library cross-reference," in the *Programmer's Guide*. An example is **strcat**, which combines (concatenates) strings:

*You can load and run this code, INTRO28.C.*

```
/* INTRO28.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <string.h>

int main()
{
   char name[60];
   strcpy(name, "Bilbo ");
   strcat(name, "Baggins");
   puts(name);

   return 0;
}
```

Here, a character array is declared, large enough to hold all strings you want. The **strcpy** function stores the first string, and then the **strcat** function adds (concatenates) the last name onto the first. (Notice that the first value of *name* ended with a space, so the names would be separated.)

## Renaming types

As you get into more complex data structures, it is helpful to be able to assign a meaningful name to them. You can do so using the **typedef** keyword. **typedef** gives a name to some combination of the standard C data types. Here are some examples:

```
#include <stdio.h>

int main()
{
   typedef unsigned char uchar;
   uchar greek_alpha = 224, greek_beta = 225;
   printf("%c %c", greek_alpha, greek_beta);
   return 0;
}
```

The **typedef** declaration gives the new name **uchar** to the type **unsigned char**. (This is a character type that can hold all 256 characters of the extended PC character set). The second statement declares *greek_alpha* and *greek_beta* to be variables of type **uchar**, and the **printf** statement displays their values.

**typedef** doesn't actually create new data types. It just makes it easier to remember what kind of data you are dealing with later in your program. As you will see, it is particularly useful for giving names to more complex data types, such as enumerations and structures.

## Enumerated types

Data sometimes fits logically into an ordered series where one item follows another—for example, the days of the week. It is convenient to use a loop to step through such values.

```
for (day = mon; day <= fri; day++)
   /* add hours worked that day to total for week */
```

Since a loop steps through numeric values, you need to give *mon* an integer value, *tues* the next integer, and so on. You could do this with **#define** statements:

```
#define mon 0
#define tues 1
#define wed 2
#define thurs 3
#define fri 4
```

However, the **enum** (enumerated) type offers a more compact way to do this, as shown in this example (load and run INTRO29.C):

```
/* INTRO29.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   enum workday {mon, tues, wed, thurs, fri};
   int day;

   for (day = mon; day <= fri; day++)
      printf("%d\n", day);
   return 0;
}
```

The first declaration automatically assigns the value 0 to *mon*, 1 to *tues*, 2 to *wed*, and so on. These names can now be used to specify the starting value and limit for a **for** loop, as shown.

Note that the numbers used in an enumeration don't have to be consecutive—you can override the default order by assigning values as follows:

```
enum scores {touchdown = 6, field_goal = 3, safety = 2, point_after = 1};
```

The statement

```
printf("%d\n", touchdown + point_after + field_goal)
```

displays the result 10.

## Combining data into structures

Arrays and enumerations give two powerful ways to handle sets of data values of the same type. A *structure* bundles together a set of data values of different types (or at least values that have different meanings). For example, information about an employee could be stored in the following structure:

```
struct employee {
   char last_name[30];
   char first_name[20];
   char initial;
   double employee_no;
   double SS_no;
   char dept_code[3];
   float annual_salary;
} ;
```

You have now defined a new data type **employee** and specified the list of data items (*members*) that a variable of **employee** type will have. As you can see, a variety of data types can be used—in this example, character arrays, single characters, doubles, and floats.

## Using parts of a structure

The next example bundles together the information that the earlier program PLANETS.C needed to draw a planet. We'll use it later to illustrate how parts of a structure can be initialized and accessed. Here they are displayed (load and run INTRO30.C):

*A field of a structure is referred to by using the structure name followed by a period and the member name. Thus `mars.distance` is the member containing the distance of Mars from the Sun in astronomical units.*

```
/* INTRO30.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <string.h>

typedef struct {
   char name[10];
   float distance;
   float radius;
} planet;

planet mars;

int main()
{
   strcpy(mars.name,"Mars");
   mars.distance = 1.5;
   mars.radius = 0.4;

   printf("Planetary statistics:\n");
   printf("Name: %s\n", mars.name);
   printf("Distance from Sun in AU: %4.2f\n", mars.distance);
   printf("Radius in Earth radii: %4.2f\n", mars.radius);

   return 0;
}
```

The **typedef** keyword gives the name *planet* to a **struct** (structure) consisting of three members, or fields: The planet name is an array of characters, while the distance and radius are floating-point values. The declaration `struct planet mars` creates a variable *mars* whose type is **planet** (in other words, it makes a copy of the structure defined earlier). In **main**, values are assigned to these three members.

As shown here, the library function **strcpy** is handy for copying a string value into structure fields that hold arrays of characters. The numeric values are simply assigned as usual.

The **printf** statements at the end of the program use the *name.member* notation to reference and print out the values that were just assigned.

## Building proper declarators

A *declarator* is a statement in C that you use to declare functions, variables, pointers, and data types. And C allows you to build very complex declarators. This section gives you some examples of declarators so that you can get some practice at designing (and reading) them; it'll also show you some pitfalls to avoid.

Traditional C programming has you build your complete declarator in place, nesting definitions as needed. Unfortunately, this can make for programs that are difficult to read (and write).

Consider, for example, the declarators in the next table, assuming that you are compiling under the small memory model (small code, small data).

Table 4.9: Declarators without typedefs

| | |
|---|---|
| `int f1();` | Function returning **int** |
| `int *p1;` | Pointer to **int** |
| `int *f2();` | Function returning pointer to **int** |
| `int far *p2;` | Far pointer to **int** |
| `int far *f3();` | Near function returning far pointer to **int** |
| `int * far f4();` | Far function returning near pointer to **int** |
| `int (*fp1)(int);` | Pointer to function returning **int** and accepting **int** parameter |
| `int (*fp2)(int *ip);` | Pointer to function returning **int** and accepting pointer to **int** |
| `int (far *fp3)(int far *ip)` | Far pointer to function returning **int** and accepting far pointer to **int** |
| `int (far *list[5])(int far *ip);` | Array of five far pointers to functions returning **int** and accepting far pointers to **int** |
| `int (far *gopher(int(far * fp[5])\` `(int far *ip)))(int far *ip);` | Near function accepting array of five far pointers to functions Returning **int** and accepting far pointers to **int**, and returning one such pointer (the backslash allows for line continuation) |

These are all valid declarators; they just get increasingly hard to understand. However, with judicious use of **typedef**, you can improve the legibility of these declarators.

Here are the same declarators, rewritten with the help of **typedef** statements:

Table 4.10: Declarators with typedefs

| | |
|---|---|
| `int      f1();` | Function returning **int** |
| `typedef int *intptr;` | |
| `intptr  p1;` | Pointer to **int** |
| `intptr  f2();` | Function returning pointer to **int** |
| `typedef int far *farptr;` | |
| `farptr  p2;` | Far pointer to **int** |
| `farptr  f3();` | Near function returning far pointer to **int** |
| `intptr  far f4();` | Far function returning near pointer to **int** |
| `typedef int (*fncptr1)(int);` | |
| `fncptr1 fp1;` | Pointer to function returning **int** and accepting **int** parameter |
| `typedef int (*fncptr2)(intptr);` | |
| `fncptr2 fp2;` | Pointer to function returning **int** and accepting pointer to **int** |
| `typedef int (far *ffptr)(farptr);` | |
| `ffptr   fp3;` | Far pointer to function returning **int** and accepting far pointer to **int** |
| `typedef ffptr   ffplist[5];` | |
| `ffplist list;` | Array of five far pointers to functions returning **int** and accepting far pointers to **int** |
| `ffptr   gopher(ffplist);` | Near function accepting array of five far pointers to functions returning **int** and accepting far pointers to **int**, and returning one such pointer |

As you can see, there's a big difference in legibility and clarity between this **typedef** declaration of *gopher* and the previous one. If you'll use **typedef** statements and function prototypes wisely, you'll find your programs easier to write, debug, and maintain.

## Pointers

Every variable has a unique memory address that indicates the beginning of the memory area occupied by its value. The amount of memory used depends on the type of data involved. In the case of an **int**, this area is 2 bytes, while a **float** uses 4 bytes. For an array, the area occupied is equal to the number of elements times the size needed for one value of the declared data type. For a structure, the area used is equal to the sum of the areas needed for the structure's members, plus some padding if needed and if you use the **–a** option. Because in all cases data is stored in an orderly, predictable way, it is possible to access data by manipulating a variable that contains the relevant address. Such a variable is called a *pointer*.

Why are pointers useful? First, they allow you to access and manipulate structured data easily, without having to move the data itself around in memory. For example, by being set to the addresses of consecutive elements in an array, a pointer can be used to initialize the array or to retrieve data from it. By adding to or subtracting from the pointer, you point to different data items.

Pointers can also be used to allow a function to receive and change the value of a variable. This can avoid the need for declaring global variables.

*Memory allocation is discussed further in Chapter 4 of the Programmer's Guide, "Memory models, floating point, and overlays."*

Also, pointers are needed for allocating memory while your program is running: In essence, you ask for a chunk of free memory (via a function such as **malloc**), and get back a pointer to the first available address.

## Declaring and using a pointer

A pointer declaration takes this form:

```
type *name
```

where *type* is any data type. Here are some example pointer declarations:

*Most programmers use names that include an abbreviation of the word pointer; for example, intptr.*

```
int *intptr;          /* Points to an integer */
float *fltptr;        /* Points to a floating-point value */
char *string;         /* Points to a character value */
```

You can declare a pointer to any object in memory, including arrays, structures, functions, and even other pointers.

To access the value pointed to by a pointer, precede the pointer name with an asterisk. For example, **intptr* yields the value stored at the address stored in the pointer *intptr*. Because this value is reached indirectly (rather than the case of a regular variable, where the value is stored at the variable's own address), this process is called *indirection* or *dereferencing*.

The following program declares a pointer and uses it to retrieve the value of a variable:

*Load and run INTRO31.C.*

```
/* INTRO31.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
```

```
{
   int intvar = 10;
   int *intptr;
   intptr = &intvar;

   printf("Location of intvar: %p\n", &intvar);
   printf("Contents of intvar: %d\n", intvar);
   printf("Location of intptr: %p\n", &intptr);
   printf("Contents of intptr: %p\n", intptr);
   printf("The value that intptr points to: %d\n", *intptr);

   return 0;
}
```

Here's the output:

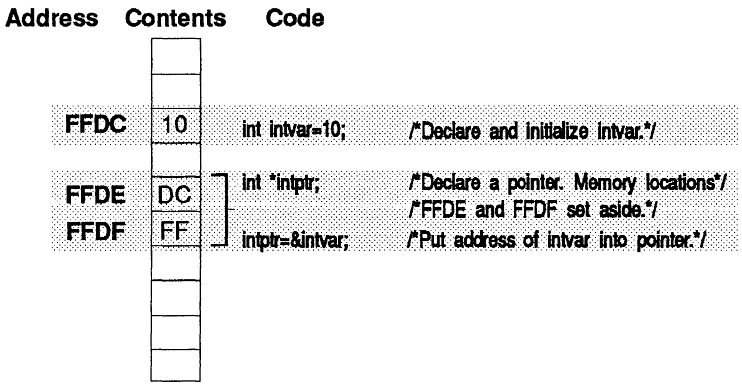
*The values of the addresses may vary.*

```
Location of intvar: FFDC
Contents of intvar: 10
Location of intptr: FFDE
Contents of intptr: FFDC
The value that intptr points to: 10
```

First, the **int** variable *intvar* is declared and assigned a value of 10. The next declaration declares an **int** * variable called *intptr*. (Read this declaration as "intptr, a pointer to an integer value.") The next statement assigns *intptr* the address of the variable *intvar* (notice the **&**, or address-of operator). A pointer must always be assigned the address of the object it is intended to point to. If you neglect this, the pointer will contain a garbage address. If you try to store something at such an address by means of the pointer, you risk destroying part of your program or data, or even halting the system.

As you can see from the output of the **printf** statements, *intvar* and *intptr* are stored at different addresses, since they are different variables. (Since they were declared consecutively, their addresses happen to be close together, but that has nothing to do with how pointers work.) As you can see, *intptr* contains the address of *intvar* (which had been assigned to it previously). The last **printf** statement prints the value pointed to by *intptr*; in other words, the contents of the address stored there. Since that address is that of *intvar*, the value pointed to is the contents of *intvar*. The next figure may help you visualize all this.

Figure 4.8
How pointers point (and what they point to)



Address
Contents
Code
FFDC
10
int intvar=10;
/*Declare and initialize intvar.*/
FFDE
DC
int *intptr;
/*Declare a pointer. Memory locations*/
/*FFDE and FFDF set aside.*/
FFDF
FF
intptr=&intvar;
/*Put address of intvar into pointer.*/

## Pointers and strings

You've seen that you can access individual characters from a string by using indexing. For example, if you declare a string *char name[20]* and store the string "Madonna" in it, the value of *name[2]* is the character *d* (remembering that the count starts at 0: *name[0]* is *M*). An alternative way to handle strings is to declare a pointer to character and use it to manipulate the string.

*This example is based on the earlier example of a string as a character array, but has been rewritten to use a pointer. To try it out, load and run INTRO32.C.*

```
/* INTRO32.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

int main()
{
   char name[40];
   char number[10];
   char *str_ptr = name;
   int pos, num_chars;

   printf("Enter a string for the character array: ");
   gets(name);
   printf("How many characters do you want to extract? ");
   gets(number);
   sscanf(number, "%d", &num_chars);

   for (pos = 0; pos < num_chars; pos++)
      printf("%c", *str_ptr++);
   printf("\n");

   return 0;
}
```

Notice that *str_ptr* is declared to be a pointer to character, and then assigned the address of the character array *name*. This could have been written in two separate statements:

```
char *str_ptr;
strptr = name;
```

but by now you know that C programmers seldom use two statements when one will do. Also notice that the address assigned is *name*, not *&name*. Referring to the name of an array (or structure) gets you the first address used to store the array's values. This is equivalent to saying `&name[0]` (the address of the first element of the array *name*); you will sometimes see the latter notation.

The rest of the program works in the same way as before, until you get to the **for** loop that extracts the requested substring. In the old version, the character being retrieved each time the loop executes was indexed from the array by using the expression **name**[*pos*]. Here, however, the pointer is used, so the reference is to the value currently being pointed at: **str_ptr++*. (The pointer is incremented each time so that it points to the next value.)

## Pointer arithmetic

Noncharacter arrays are handled with pointers in the same way as strings are. You increment the pointer to point to the next element in the array; you decrement it to point to the previous item. You have to check, of course, to make sure that you aren't pointing to a location outside the bounds of the array. You usually do this by setting the appropriate limit for the loop statement used.

It is important to remember that when the pointer *ptr* is incremented, it does *not* necessarily point to the next address—in fact, it usually doesn't. The distance between addresses pointed to is equal to the size of the data type to which the pointer points. An **int** pointer points two addresses ahead when it's incremented; a **double** pointer points eight addresses ahead. C handles this automatically.

## Pointers, structures, and lists

You may remember that you get the value of a member field of a structure by using the notation *structure_name.member_name*, so that the salary field in the employee structure named *jim* would be *jim.salary*. How do you access structures and parts of structures with a pointer? The following example shows how (load and run SOLAR.C):

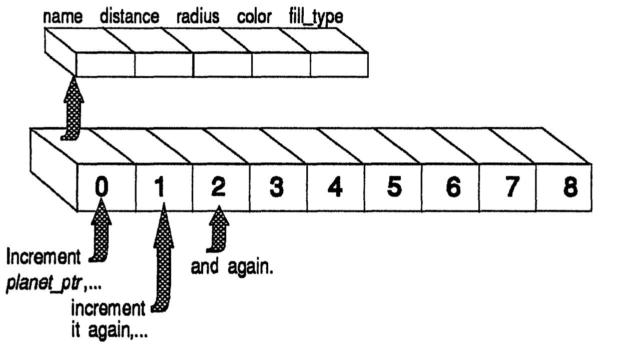
*Figure 4.9 illustrates how this program works.*

```
/* SOLAR.C--Example from Chapter 4 of Getting Started */

#include <graphics.h>
#include <stdio.h>
#include <string.h>

typedef struct {
   char name[10];
   float distance;
   float radius;
   int color;
   int fill_type;
} planet;

planet solar_system[9];
planet *planet_ptr;
int    planet_num;

int main()
{
   strcpy(solar_system[0].name,"Mercury");
   solar_system[0].distance = 0.4;
   solar_system[0].radius = 0.4;
   solar_system[0].color = EGA_YELLOW;
   solar_system[0].fill_type = EMPTY_FILL;

   planet_ptr = solar_system;
   planet_ptr++;            /* Point to second planet structure */
    strcpy (planet_ptr->name,"Venus");
   planet_ptr->distance = 0.7;
   planet_ptr->radius = 1.0;
   planet_ptr->color = EGA_BROWN;
   planet_ptr->fill_type = SOLID_FILL;

   planet_ptr = solar_system;         /* Reset to first element */
   for (planet_num = 0; planet_num < 2; planet_num++, planet_ptr++) {
      printf("\nPlanetary statistics:\n");
      printf("Name: %s\n", planet_ptr->name);
      printf("Distance from Sun in AU: %4.2f\n",
              planet_ptr->distance);
      printf("Radius in Earth radii: %4.2f\n", planet_ptr->radius);
      printf("Color constant value %d\n", planet_ptr->color);
      printf("Fill pattern constant value %d\n",
             planet_ptr->fill_type);
   }

   return 0;
}
```

*The -> notation is explained on page 114.*

The following figure illustrates how the previous code might "look" during the first few steps through its **for** loop:

Figure 4.9
Using pointers to access an array of structures



**planet solar_system[9]**
**Array of nine planet structures**

The *planet* structure is an expanded version of the one shown earlier, with members added for the color and fill type. The next declarations

```
planet solar_system[9];
planet *planet_ptr;
```

specify an array, *solar_system*, whose nine members are copies of the *planet* structure and *planet_ptr*, a pointer to a *planet* structure.

The first group of statements in **main** initialize the members of the first planet structure, using array and structure notation. To refer to a member of a structure in an array of structures, use the form

```
array_name[index].member_name
```

Thus the distance of the fourth planet in *solar_system* can be referred to as `solar_system[3].distance`.

*Since planet_ptr was declared as a pointer to structure type **planet**, the built-in pointer arithmetic takes care of moving the pointed-to address enough bytes to reach the next element of solar_system.*

The second block of statements initialize a *planet* structure using pointers. Before you can use a pointer, you must assign it a valid address. The statement `planet_ptr = solar_system` sets *planet_ptr* to point to the first element of the array *solar_system*. This is the element that has just been initialized, containing information for the planet Mercury. Therefore, the statement `planet_ptr++` points to the next (second) element.

With the pointer properly pointed, information about the second planet (Venus) is assigned to the second element of *solar_system*.

Here, however, rather than array index notation, you see the use of the pointer *planet_ptr* to access the members of the *planet* structure. This takes the form

```
pointer_name->member_name
```

so that, for example, the *distance* member for the *planet* structure currently being pointed to is `planet_ptr->distance`.

The rest of the program displays the two planet structures that have been initialized. First, *planet_ptr* is set back to the first element by being assigned the address *solar_system*. The **for** loop increments *planet_ptr* and obtains the structure's member values using the notation just discussed. Notice that the reference `planet_ptr->distance` is a bit less cumbersome than

```
solar_system[index].distance
```

where *index* is the current element number.

In fact, for any array, the index of an array is actually a pointer to the address of the array plus the index value, which internal pointer arithmetic converts to the array address plus

```
index * sizeof(type)
```

where *type* is the declared type of the array, and **sizeof** is a C operator that returns the number of bytes used by a type or variable.

## Using pointers to return values from functions

Pointers also allow you to change the actual value of the variable or variables used in calling a function. So far, functions have been called only with constant values or the *names* of variables. When you make a call such as `draw(x_cor, y_cor, size, color)`, you are passing the *values* of the specified variables to the function **draw**. The function actually gets *copies* of these values and can refer to them by name and manipulate them, but this has no effect on the actual variables used by the caller.

Sometimes it is useful to be able to call a function using variable names and have the function actually change the values of the variables themselves. This function is an example:

```
void swap(int *a, int *b)
```

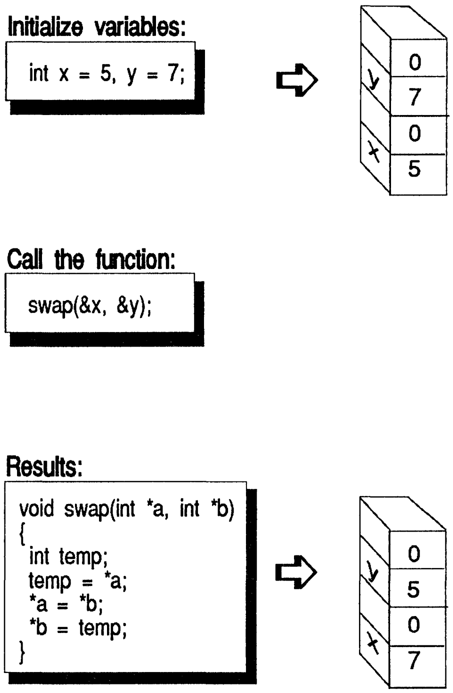
**swap** swaps the values of the two variables with which the function is called. In order for the function to access the variables themselves, however, it must have not their values but their *addresses*, so it can write the new values back to their location in memory. Therefore, the parameters are pointers to the appropriate variables, not the variables themselves. Since a pointer refers to an address, you would call this function to swap the variables *x* and *y* using the statement `swap(&x, &y)`.

Here is the definition for the **swap** program, together with a **main** function that tests it (load and run INTRO33.C).

```
/* INTRO33.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>

void swap(int *, int *);    /* This is swap's prototype */

int main()
{
   int x = 5, y = 7;
   swap(&x, &y);
   printf("x is now %d and y is now %d\n", x, y);

   return 0;
}

void swap(int *a, int *b)    /* swap is actually defined here */
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Notice that the **swap** function uses indirection (the * notation) to refer to the *values* contained in the variables *x* and *y*. The value of *x* is first stored in a temporary variable, then the value of *y* is stored in *x*. The former value of *x* is then obtained from the temporary variable and stored in *y*.

Figure 4.10
Using pointers in a function



```
int x = 5, y = 7;
```

```
swap(&x, &y);
```

```
void swap(int *a, int *b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
}
```

When you use pointers, a function is not limited to just returning one value via a **return** statement. A function can change the values of any variables it is given access to. While this could also be achieved (in this example) by making *a* and *b* global variables (declaring them outside of a function definition), it is easy to accidentally change a global variable. Pointers keep the transaction private.

## Using system resources

Thus far, the example programs have operated nearly in a vacuum. Some programs read data that you typed at the keyboard, and every program displayed something on the screen. None of the data was stored in permanent form—if you want the data back, you must run the program again. In real applications, programs usually have to read in the bulk of their data from the disk drive, a communications port, or some other source. When the data has been processed, the program may need to send it to the printer or write it to a disk file for later use. This is true of word processors, spreadsheets, and databases, for example.

This section shows you how you can use Turbo C++ to read data from or write data to a disk file. Conceptually, Turbo C++ sets up something called a *stream* through which the data moves. There are lower-level ways to work with MS-DOS files, and library functions for dealing with them, but the stream features are recommended for portability. They allow you to use files without worrying about the target machine's operating system.

A stream represents a file on the disk or some other device from which data can be read or to which it could be sent. (Many devices are used for input or output but not both—you can't usually read data from a printer, and data sent to a keyboard won't accomplish much.) In your program, you don't manipulate the stream directly. Instead, the library defines a variable of the **struct** type *FILE* (which is defined in stdio.h). Since this structure manages a memory buffer for the file contents, you then declare a pointer to your *FILE* variable and use it to manipulate the file.

To open a stream, you use a function (usually **fopen**) from the run-time library. You specify whether you want to read from the stream, write to the stream, or both. You also indicate whether you will treat the stream as text or binary data.

Text streams are used for normal DOS files. The standard stream I/O assumes that text files consist of lines of text that each end with a single newline character (the ASCII linefeed character). DOS, however, stores files with both a carriage return and a newline character following each line. When you declare a text stream, Turbo C++ translates CR/LF to a single linefeed on input, but translates the single linefeed to CR/LF when storing files on disk, maintaining compatibility with DOS.

## Using files and streams

The basic steps involved in using a disk file with your Turbo C++ program are as follows

1. Use #**include** <stdio.h> to include the necessary declarations for using files.
2. Declare a pointer to the type *FILE* (defined in stdio.h).

3. Declare the data object(s) that will be used to receive information from the file. For example, a character array to hold a line of text, an array of some numeric type, or an array of structures.
4. Open the file using **fopen**, including the file name and the type of access you will need (read only, writing, appending, and so on. See Chapter 2, "Run-time library cross-reference," in the *Programmer's Guide* for more about file access functions).
5. Make sure the file was opened by checking the value returned by **fopen**.
6. Use the appropriate library function(s) to write or read the data. For text data, **fprintf** and **fscanf** work the same as their screen-oriented counterparts **printf** and **sscanf**. **fputc** and **fgetc** deal with single characters. For reading structured blocks of data of known size, **fwrite** and **fread** may be more suitable.
7. When you're done, close the file by using **fclose**.

The following program opens a file, saves three lines of text in it, then reads them back out:

*Load and run INTRO34.C.*

```
/* INTRO34.C--Example from Chapter 4 of Getting Started */

#include <stdio.h>
#include <stdlib.h>

FILE *textfile;           /* Pointer to file being used */
char line[81];            /* Char array to hold lines read from file */

int main()
{
   /* Open file, testing for success */
   if ((textfile = fopen("intro34.txt", "w")) == NULL) {
      printf("Error opening text file for writing\n");
      exit(0);
   }

   /* Write some text to the file */
   fprintf(textfile, "%s\n", "one");
   fprintf(textfile, "%s\n", "two");
   fprintf(textfile, "%s\n", "three");

   /* Close the file */
   fclose(textfile);

   /* Open file again */
   if ((textfile = fopen("intro34.txt", "r")) == NULL) {
      printf("Error opening text file for reading\n");
      exit(0);
   }
   /* Read file contents */
   while ((fscanf(textfile, "%s", line) != EOF))
      printf("%s\n", line);

   /* Close file */
   fclose(textfile);

   return 0;
}
```

## Opening a stream

You have actually been using files all along. Every C program has automatic access to five streams, as shown in the next table.

Table 4.11
Preopened streams in Turbo C++

| Name | Function | Connected to |
|---|---|---|
| stdin | Standard input | Keyboard |
| stdout | Standard output | Screen |
| stderr | Standard error | Screen |
| stdaux | Standard auxiliary | Serial port |
| stdprn | Standard print | Printer port |

The first two default streams can be redirected in various ways. Due to their default connections, your Turbo C++ programs expect to get input from the keyboard and send their output to the screen. When you open a file with a pointer to the *FILE* type, you are opening an additional stream.

In the example program, a pointer *textfile* to a stream is declared. The **fopen** statement associates this pointer with the disk file named *text*, which is opened with the access mode **w** (for write access). The **if** statement checks for the return value NULL, which is a predefined pointer indicating failure to open the file.

## Writing to the file

**fprintf** writes the three lines of text to the file. **fprintf** works just like the familiar **printf** except that it gives the name of the stream pointer first, then the format specifier, and then the data to be written. Notice that a new line (**\n**) must be used to separate the lines of text so that they can be read later with **fscanf**. After the data is written, the file is closed with **fclose**.

## Reading from the file

To read the text back from the file, the file is opened again with **fopen**, this time in read mode. **fscanf** reads each line into the character array *line*. A regular **printf** statement displays the line on the screen. The whole thing goes into a **while** loop that checks for **fscanf** to return the predefined value EOF (end of file).

C H A P T E R

5

# A C++ primer

*This chapter covers the basic ideas of C++; Chapter 6, "Hands-on C++," takes you on a rapid romp through several C++ program examples.*

This chapter gives you the feel and flavor of the C++ language. We demystify some of the jargon and combine a little theory with simple, illustrative programs. The source code for these examples is provided on your distribution diskettes so you can study, edit, compile, and run them. (The graphics examples, of course, will run only if you have a graphics adapter and monitor. Any CGA, EGA, VGA, or Hercules setup will do.)

Turbo C++ provides all the features of AT&T's C++ version 2.0. C++ is an extension of the popular C language, adding special features for object-oriented programming (OOP).

OOP is a method of programming that seeks to mimic the way we form models of the world. To cope with the complexities of life, we have evolved a wonderful capacity to generalize, classify, and generate abstractions. Almost every noun in our vocabulary represents a class of objects sharing some set of attributes or behavioral traits. From a world full of individual dogs, we distill an abstract class called *dog*. This allows us to develop and process ideas about canines without being distracted by the details concerning any particular dog. The OOP extensions in C++ exploit this natural tendency we have to classify and abstract things—in fact, C++ was originally called "C with Classes."

Three main properties characterize an OOP language:

- *Encapsulation*: Combining a data structure with the functions (actions or *methods*) dedicated to manipulating the data.

Encapsulation is achieved by means of a new structuring and data-typing mechanism—the *class*.

- *Inheritance*: Building new, *derived* classes that inherit the data and functions from one or more previously-defined *base* classes, while possibly redefining or adding new data and actions. This creates a *hierarchy* of classes.
- *Polymorphism*: Giving an action one name or symbol that is shared up and down a class hierarchy, with each class in the hierarchy implementing the action in a way appropriate to itself.

Borland's C++ gives you the full power of object-oriented programming:

- more control over your program's structure and modularity
- the ability to create new data types with their own specialized operators
- and the tools to help you create reusable code

All these features add up to code that can be more structured, extensible, and easier to maintain than that produced with non-object-oriented languages.

To achieve these important benefits of C++, you may need to modify ways of thinking about programming that have been considered standard for many years. Once you do that, however, C++ is a simple, straightforward, and superior tool for solving many of the problems that plague traditional software.

Your background may affect the way you look at C++:

***If you are new to C and C++.*** You may at first have some difficulty with the new concepts discussed in this chapter, but working through (and experimenting with) the examples will help make the ideas concrete. Before you begin, you should make sure you understand the basic elements of the C language (you may wish to review Chapter 4 before continuing here). As a beginner, you have one very real advantage: You probably have fewer old programming habits to unlearn.

***If you are an experienced C programmer.*** C++ builds upon the existing syntax and capabilities of C. This makes the learning curve much gentler than if you had to learn a whole new language. It also allows you to port existing C programs to C++ with a minimum of recoding. You aren't losing C's power and efficiency: You're adding the representational power of classes and the security of controlling access to internal data.

***If you program in Turbo Pascal 5.5.*** Turbo Pascal 5.5 embodies many of the same object-oriented features found in C++. While you will have to deal with basic syntax differences between the two languages, you will find that Turbo Pascal 5.5 *objects* and Turbo C++ *classes* are structured similarly. You will recognize C++ *member functions* as being like Turbo Pascal 5.5's *methods,* and may note many other similarities. The main difference you will observe is that C++ has tighter control over data access.

***If you are experienced in another object-oriented programming language.*** You will find some differences in C++:

- First, the syntax of C++ is that of a traditional, procedural language.
- Second, the way C++ and Smalltalk actually deal with objects during compilation is different. Smalltalk's binding is done completely at run time (*late binding*); C++ allows both compile-time (early) binding and late binding.

In this chapter, we begin by describing the three key OOP ideas—encapsulation, inheritance, and polymorphism—in more detail. The first listings show fragments of code to illustrate each topic. Later, we present complete, compilable programs. The main example develops object-oriented representations useful for graphics, but occasional side tours show how C++ works with strings and other data structures.

## Encapsulation

How does C++ change the way you work with code and data? One important way is *encapsulation*: the welding of code and data together into a single class-type object. For example, you might have developed a data structure, such as an array holding the information needed to draw a character font on the screen, and code (functions) for displaying, scaling and rotating, highlighting, and coloring your font characters.

In traditional C, the usual solution is to put the data structures and related functions into a single separately compiled source file in an attempt to treat code and data as a single module. While this is a step in the right direction, it isn't good enough. There is no explicit relationship between the data and the code, and you or another programmer can still access the data directly without using the functions provided. This can lead to problems. For example, suppose that you decide to replace the array of font information with a linked list? Another programmer working on the same project may decide that she has a better way to access the character data, so she writes some functions of her own that manipulate the array directly. The problem is that the array isn't there any more!

C++ comes to the rescue by extending the power of C's **struct** and **union** keywords, and by adding a keyword not found in C: **class**. All three keywords are used in C++ to define *classes*.

*In these manuals, we use bold type to distinguish the keyword **class** from the generic word "class."*

In C++, a single class entity (defined with **struct, union,** or **class**) combines functions (known as *member functions*) and data (known as *data members*). You usually give a class a useful name, such as **Font**. This name becomes a new type identifier that you can use to declare *instances* or *objects* of that class type:

```
   class Font {
// here you declare your members: both data and functions;
// don't worry how for the moment.
   };
   Font Tiffany;         // declares Tiffany to be of type class
                         // Font.
```

Note that in Turbo C++ you can now use two slashes (//) to introduce a single-line comment in both C and C++. You can still use the /* */ comment characters if you prefer them; in fact, they are especially useful for long comments.

***Warning!*** Use of the // comments is not usually portable to other C compilers. However, it is portable to other C++ compilers.

The variable *Tiffany* is an *instance* (sometimes called an *instantiation*) of the class **Font**. You can use the class name **Font** very much like a normal C data type. For example, you can declare arrays and pointers:

```
Font Times[10];     // declare an array of 10 Fonts
Font* font_ptr;     // declare a pointer to Font
```

A major difference between C++ classes and C structures concerns the accessibility of members. The members of a C structure are freely available to any expression or function within their scope. With C++, you can control access to **struct** and **class** members (code and data) by declaring individual members as

**public**, **private**, or **protected**. (A C++ union is more like a C union, with all members **public**.) We'll explain these three access levels in more detail later on.

*C++ **structs** and **unions** are not quite the same as the C versions.*

C++ structures and unions offer more than their C counterparts: they can hold function declarations and definitions as well as data members. In C++, the keywords **struct**, **union**, and **class** can all be used to define classes.

- A class defined with **struct** is simply a class in which all the members are **public** by default (but you can vary this arrangement if you wish).
  A class defined with **union** has all its members **public** (this access level cannot be changed).
- In a class defined with **class**, the members are **private** by default (but there are ways of changing their access levels).

So, when we talk about classes in C++, we include structures and unions, as well as types defined with the keyword **class**.

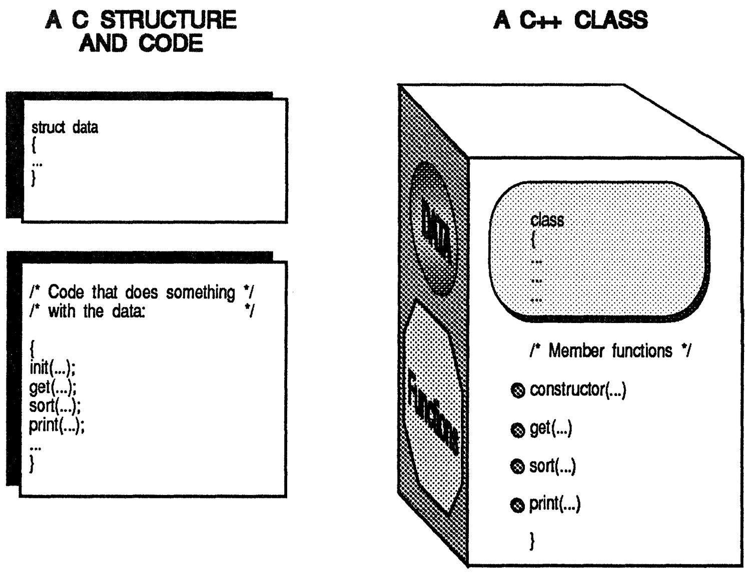
Typically, you restrict member-data access to member functions: you usually make the member data **private** and the member functions **public**.

Returning to the problem of handling fonts, how does the C++ class concept help?

By creating a suitable **Font** class, you can ensure that the private font data can be accessed and manipulated *only* through the public **Font** member functions that you have created for that purpose. You are now free at any time to change the font data structure from an array to a linked list, or whatever. You would, of course, need to recode the member functions to handle the new font data structure, but if the function names and arguments are unchanged, programs (and programmers) in other parts of your system will be unaffected by your improvements!

The next figure compares the ways C and C++ provide access to a font.

Figure 5.1
Traditional C versus
encapsulated C++



```
struct data
{
...
}
```

```
/* Code that does something */
/* with the data:           */

{
init(...);
get(...);
sort(...);
print(...);
...
}
```

```
class
{
...
...
...

/* Member functions */
● constructor(...)
● get(...)
● sort(...)
● print(...)
}
```

Thus the technique of encapsulation in classes helps provide the very real benefit of *modularity*, as found in languages such as Ada and Modula-2. The C++ class establishes a well-defined interface that helps you design, implement, maintain, and reuse programs. Debugging a C++ program is often simpler since many errors can be quickly traced to one particular class.

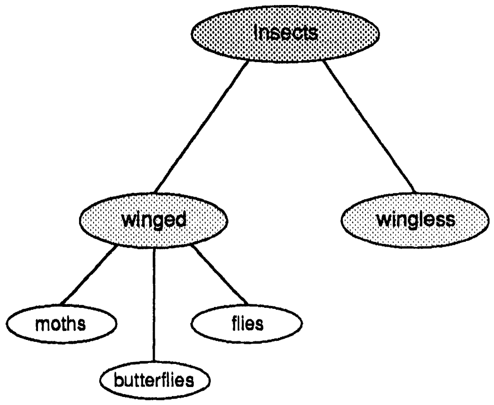
The class concept leads to the idea of *data abstraction*. Our font data structure is no longer tied to any particular physical implementation; rather, it is defined in terms of the operations (member functions) allowed on it. At the same time, the traditional C philosophy that views a program as a collection of *functions*, with data as second-class citizens, has also shifted. The C++ class weds data and function as equal, interdependent partners.

## Inheritance

The descriptive branches of science (required before the explanatory and predictive aims of science can bear fruit) spend much time classifying objects according to certain traits. It often helps to organize your classification as a family tree with a single overall category at the root, with subcategories branching out into subsubcategories, and so on.

Entomologists, for example, classify insects as shown in Figure 5.2. Within the phylum *insect* there are two divisions: winged and wingless. Under winged insects is a larger number of categories: moths, butterflies, flies, and so on.

Figure 5.2
A partial taxonomy chart of insects



This classification process is called *taxonomy*. It's a good starting metaphor for OOP's inheritance mechanism.

The questions we ask in trying to classify some 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. We begin at the top of a specimen's family tree and start 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.

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 *fly* inherits that characteristic from its order.

OOP is the process of building class hierarchies. One of the important things C++ adds to C is a mechanism by which class types can inherit characteristics from simpler, more general types. This mechanism is called *inheritance*. Inheritance provides for commonalty of function while allowing as much specialization as needed. If a class **D** inherits from class **B**, we say that **D** is the *derived* class and **B** is the *base* class.

It is by no means a trivial task, though, to establish the ideal class hierarchy for a particular application. The insect taxonomy took hundreds of years to develop, and is still subject to change and acrimonious debate. Before you write a line of C++ code, you must think hard about which classes are needed at which level. As the application develops, you may find that new classes are required that fundamentally alter the whole class hierarchy. The bibliography lists many books on this subject. Remember also that a growing number of vendors are supplying Turbo C++ compatible libraries of classes. So don't reinvent too many wheels.

Occasionally, you encounter a class that combines the properties of more than one previously established class. C++ version 2.0 offers a mechanism (not found in earlier C++ versions) known as *multiple inheritance* whereby a derived class can inherit from two or more base classes. You'll see later how this is achieved as a logical extension of the single inheritance mechanism.

## Polymorphism

The word *polymorphism* comes from the Greek: "having many shapes." Polymorphism in C++ is accomplished with *virtual* functions. Virtual functions let you use many versions of the same function throughout a class hierarchy, with the particular version to be executed being determined at run time (this is called *late binding*).

## Overloading

In C, you can only have one function with a given name. For example, if you declare and define the function

```
int cube (int number);
```

you can now get the cube of an integer. But suppose you want to cube a **float** or a **double**? You can of course declare functions for these purposes, but they can't use the name **cube**:

```
float fcube (float float_number);
double dcube (double double_number);
```

In C++, however, you can *overload* functions. This means that you can have several functions that have the same name but work with different types of data. Thus you can declare:

```
int cube (int number);
float cube (float float_number);
double cube (double double_number);
```

As long as the argument lists are all different, C++ takes care of calling the correct function for the argument given. If you have the call **cube**(10); the **int** version of **cube** is called, while if you call **cube**(2.5); the **double** version will be called. If you call **cube**(2.5F), then you are passing a floating-point literal rather than a **double,** and the **float** version will be called. Even operators such as **+** can be overloaded and redefined so they work not only with numbers, but with graphic objects, strings, or whatever is appropriate for a given class.

## Modeling the real world with classes

The C++ class provides a natural way of building computer models of real-world systems—indeed, Bjarne Stroustrup devised the language at AT&T Bell Labs in order to model a large telephone switching system.

There have been many C++ applications in the motor industry. When modeling vehicles, for instance, you would be interested in both the physical description (the number of tires, engine power, weight, and so on) and the behavior (acceleration, breaking, steering, fuel consumption). A **Car** class could encapsulate the physical parameters (data) and their behavior (functions) in a very general way. Using inheritance, you might then derive specialized **Sports_car** and **Station_wagon** classes, adding new data types and functions, as well as modifying (overriding) some of the functions of the base class. Much of the coding you have done for the base class(es) is reused or at least recycled.

### Building classes: a graphics example

In a graphics environment, a reasonable place to start would be a class that models the physical pixels on a screen with the abstract points of plane geometry. A first try might be a **struct** class called **Point** that brings together the *X* and *Y* coordinates as data members:

```
struct Point { // defines a struct class called Point
   int X;      // struct member data are public by default
   int Y;
};
```

*When you define a class, you add a new data type to C++. The language treats your new data type in the same way that it treats built-in data types.*

You can now declare several particular variables of type **struct Point** (for brevity, we often loosely refer to such variables as being of type **Point**). In C, you would use declarations such as

```
struct Point Origin, Center, Cur_Pos, AnyPoint;
```

but in C++, all you need is

```
Point Origin, Center, Cur_Pos, AnyPoint;
```

*The terms object and class instance are used interchangeably in C++.*

A variable of type **Point** (such as *Origin*) is one of many possible instances of type **Point**. Note carefully that you assign values (particular coordinates) to *instances* of the class **Point**, not to **Point** itself. Beginners often confuse the data type **Point** with the instance variables of type **Point**. You can write `Center = Origin` (assign *Origin*'s coordinates to *Center*), but `Point = Origin` is meaningless.

When you need to think of the *X* and *Y* coordinates separately, you can think of them as independent members (fields) *X* and *Y* of the structure. On the other hand, when you need to think of the *X* and *Y* coordinates working together to fix a place on the screen, you can think of them collectively as **Point**.

Suppose you want to display a point of light at a position described on the screen. In addition to the *X* and *Y* location members you have already seen, you'll want to add a member that specifies whether there is an illuminated pixel at that location. Here's a new **struct** type that includes all three members:

*The Boolean type will be familiar to Turbo Pascal programmers.*

```
enum Boolean {false, true};   // false = 0; true = 1

struct Point {
   int X;
   int Y;
   Boolean Visible;
};
```

This code uses an enumerated type (**enum**) to create a true/false test. Since the values of enumerated types start at 0, *Boolean* can have one of two values: 0 or 1 (false or true).

## Declaring objects

As with other data types, you can have pointers to classes and arrays of classes:

```
Point Origin;        // declare object Origin of type Point
Point Row[80];       // declare an array of 80 objects of type Point
Point *point_ptr;    // declare a 'pointer to type Point'
point_ptr = &Origin; // point it to the object Origin
point_ptr = Row;     // then point it to Row[0]
```

## Member functions

As you saw earlier, C++ classes can contain functions as well as data members. A *member function* is a function declared *within* the class definition and tightly bonded to that class type. (Member functions are known as *methods* in other object-oriented languages, such as Turbo Pascal and Smalltalk.)

*Data members are what the class **knows**; its member functions are what the class **does**.*

Let's add a simple member function, **GetX**, to the class *Point*. There are two ways of adding a member function to a class:

- Define the function inside the class
- Declare it inside the class, then define it outside the class

The two methods have different syntaxes and technical implications.

The first method looks like this:

```
struct Point {
   int X, Y;
   Boolean Visible;
   int GetX() { return X;}  // inline member function defined
};
```

*Inline functions are discussed in more detail on pages 139 and 175.*

This form of definition makes **GetX** an *inline* function by default. Briefly, inline functions are functions "small" enough to be usefully compiled *in situ*, rather like a macro, avoiding the overhead of normal function calls.

Note that the inline member function definition follows the usual C syntax for a function definition: the function **GetX** returns an **int** and takes no arguments. The body of the function, between **{** and **}**, contains the statements defining the function—in our case, the single statement, `return X;`.

In the second method, you simply *declare* the member function within **struct Point**, (using normal C function declaration syntax), then provide its full *definition* (complete with the body statements) elsewhere, outside the body of the class definition.

```
struct Point {
   int X, Y;
   Boolean Visible;
   int GetX();      // member function declared
};

   int Point::GetX() { // member function defined
      return X;        // outside the class
   }
```

*The :: is known as the scope resolution operator; it tells the compiler where the function belongs.*

Member functions defined outside the class definition still can be made inline (if certain conditions are met), but you have to request this explicitly with the keyword **inline**.

Note carefully the use of the scope resolution operator in **Point::GetX** in the function definition. The class name **Point** is needed to tell the compiler which class **GetX** belongs to (there may be other versions of **GetX** around belonging to other classes). The inside definition did not need the **Point::** modifier, of course, since that **GetX** clearly belongs to **Point.**

*Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide explains class scope in more detail.*

The **Point::** in front of **GetX** also serves another purpose. Its influence extends into the function definition, so that the *X* in `return X;` is taken as a reference to the *X* member of the class **Point**. Note also that the body of **Point::GetX** is within the scope of **Point** regardless of its physical location.

Whichever defining method we use, the important point is that we now have a member function **GetX** tied to the class **Point.** Since it is a member function, it can access all the data variables that belong to **Point**. In our simple case, **GetX** just accesses *X*, and returns its value.

## Calling a member function

Now member functions represent operations on objects of their class, so when we call **GetX** we must somehow indicate which **Point** object is being operated on. If **GetX** were a normal C function (or a C++ nonmember function), this problem would not arise—you would simply invoke the function with the expression, **GetX()**. With member functions, you must supply the name of the target object. The syntax used is a natural extension of that used in C to reference structure members. Just as you would refer to *Origin.X* for the *X* component of the object *Origin*, or to *Endpoint.Y* for the *Y* component of the object *Endpoint*, you can invoke **GetX** with **Origin.GetX()** or **Endpoint.GetX()**. The "." operator serves as the class component selector for both data and function members. The general calling syntax is

*class-object-name.member-function-name(argument-list)*

In the same way, if you had a pointer to a **Point** object, you would use the pointer member selector, **"->"**: `Point_pointer->GetX()`. You'll see many examples of such member function calls in the examples in this chapter.

## Constructors and destructors

There are two special types of member functions, *constructors* and *destructors*, that play a key role in C++. To appreciate their importance, a short detour is needed. A common problem with traditional languages is *initialization*: Before using a data structure, it must be allocated memory and initialized. Consider the task of initializing the structure defined earlier:

```
struct Point {
   int X;
   int Y;
   Boolean Visible;
};
```

Inexperienced programmers might try to assign initial values to the *X*, *Y*, and *Visible* members in the following way:

```
Point ThisPoint;
ThisPoint.X = 17;
ThisPoint.Y = 42;
ThisPoint.Visible = false;
```

This works, but it's tightly bound to one specific object, *ThisPoint*. If more than one *Point* object needs to be initialized, you'll need more assignment statements that do essentially the same thing. The natural next step is to build an initialization function that generalizes the assignment statements to handle any **Point** object passed as an argument:

```
void InitPoint(Point *Target, Int NewX, Int NewY)
{
   Target->X = NewX;
   Target->Y = NewY;
```

```
        Target->Visible = false;
    }
```

This function takes a pointer to a **Point** object and uses it to assign the given values to its members (note again the **->** operator when using pointers to refer to class members). You've correctly designed the function **InitPoint** specifically to serve the structure **Point**. Why, then, must you keep specifying the class type and the particular object that **InitPoint** acts upon? The answer is that **InitPoint** is not a member function. What we really need for true object-oriented bliss is a member function that will initialize any **Point** object. This is one of the roles of the constructor.

C++ aims to make user-defined data types as integral to the language (and as easy to use) as built-in types. Therefore, C++ provides a special type of member function called a *constructor*. A constructor specifies how a new object of a class type will be created, i.e., allocated memory and initialized. Its definition can include code for memory allocation, assignment of values to members, conversion from one type to another, and anything else that might be useful. Constructors can be user-defined, or C++ can generate default constructors. Constructors can either be called explicitly or implicitly. The C++ compiler automatically calls the appropriate constructor whenever you define a new object of the class. This can happen in a data declaration, when copying an object, or through the dynamic allocation of a new object using the operator **new**.

*Destructors*, as the name indicates, destroy the class objects previously created by a constructor by clearing values and deallocating memory. As with constructors, destructors can be called explicitly (using the C++ operator **delete**) or implicitly (when an object goes out of scope, for example). If you don't define a destructor for a given class, C++ generates a default version for you. Later on, we'll be looking at the syntax for defining destructors. First, though, let's see how constructors are made.

The following version of **Point** adds a constructor:

```
struct Point {
   int X;
   int Y;
   Boolean Visible;
   int GetX() {return X;}
   Point(int NewX, int NewY); // constructor declaration
};
```

*Point::**Point** indicates that we are defining a constructor for the class Point.*

```
Point::Point(int NewX, int NewY) // constructor definition
{
   X = NewX;
   Y = NewY;
   Visible = false;
};
```

The constructor definition here is made *outside* the class definition. Constructors can also be legally defined *inside* the class, as inline functions. Or they can be defined outside the class definition and made inline with the keyword **inline**. However, some care is needed: the amount of code generated by a constructor is not always proportional to the visible source code in its definition.

Notice that the name of a constructor is the same as the name of the class: **Point**. That's how the compiler knows that it is dealing with a constructor. Also note that a constructor can have arguments as with any other kind of function. Here the arguments are *NewX* and *NewY*. The constructor body is built just like the body of any member function, so a constructor can call any member functions of its class or access any member data. A constructor, though, *never* has a return type—not even **void**.

Now you can declare a new **Point** object like this:

```
Point Origin(1,1);
```

This declaration invokes the previously defined **Point** constructor for you. As you'll see later, you can have more that one constructor for a class—and, as with other C++ overloaded functions, the appropriate version will be automatically invoked according to the argument lists involved. You'll also see that if you do not define a constructor, C++ generates a default constructor with no arguments.

Another useful trick in C++ is that you can have default values for function arguments:

```
Point::Point(int NewX=0, int NewY=0) // revised constructor
definition
{
// as before
}
```

The declaration,

```
Point Origin(5);
```

would initialize *X* to *5* and *Y* to *0* by default.

## 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 exist 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 function with the wrong data or the wrong function with the right data. Matching the two is the programmer's job, and while ANSI C, unlike traditional C, provides good type-checking, at best it can only say what *doesn't* go together.

By bundling code and data declarations together, C++ classes help keep them in sync. Typically, to get the value of one of a class's data members, you call a member function belonging to that class which returns the value of the desired member. To set the value of a field, you call a member function that assigns a new value to that field.

## Member access control: private, public, and protected

While the enhanced **struct** in C++ allows bundling of data and functions, it is not as encapsulated or modular as it could be. As we mentioned earlier, access to all data members and member functions of a **struct** is **public** by default—that is, any statement within the same scope can read or change the internal data of a **struct** class. As noted earlier, this isn't desirable and can lead to serious problems. Good C++ design practices *data hiding* or *information hiding*—keeping member data private or protected, and providing an authorized interface for accessing it. The general rule is to make all data private so that it can be accessed only through public member functions. There are only a few situations where public rather than private or protected data members are needed. Also, some member functions involved only in internal operations can be made private or protected rather than public.

Three keywords provide access control to structure or class members. The appropriate keyword (with a colon) is placed before the member declarations to be affected:

| | |
|---|---|
| **private:** | Members following this keyword can be accessed only by member functions declared within the same class. |
| **protected:** | Members following this keyword can be accessed by member functions within the same class, and by member functions of classes that are derived from this class (see the discussion on page 141). |
| **public:** | Members following this keyword can be accessed from anywhere within the same scope as the class definition. |

For example, here is how to redefine the **Point** structure so that the data members are private and the member functions are public:

```
struct Point {
private:
   int X;
   int Y;

public:
   int GetX();
   Point(int NewX, int NewY);
};
```

## The class: private by default

A **struct** class is public by default, so you have to use **private:** to specify the private part, and then **public:** for the part to be made available for general access. Since good C++ practice makes things private by default and carefully specifies what should be public, C++ programmers generally favor the **class** over the **struct**. The only difference between a **class** and a **struct** is this matter of default privacy.

**Point** redefined as a class looks like this:

```
class Point {
   int X;        // private by default
   int Y;

public:          // needed to override the private default
   int GetX();
   Point(int NewX, int NewY);
};
```

No **private** modifier is needed for the data members—they're **private** by default. The member functions, however, must be declared **public** so that they can be used outside of the class to initialize and retrieve values of **Point** objects.

You can repeat access control specifications as often as needed:

```
enum Boolean {false, true};

class Employee {
   double salary;              // private by default
   Boolean permanent;
   Boolean professional;

public:
   char name[50];
   char dept_code[3];

private:
   int Error_check(void);

public:
Employee(double salary, Boolean permanent, Boolean professional,
         char *name, char *dept_code);
};
```

*Data members are usually* ***private****, while member functions are usually* ***public****. Allow public access only where it is truly needed.*

Here the data members *salary, permanent,* and *professional* are **private** by default; the data members *name* and *dept_code* are declared to be **public**; the member function **Error_check** is declared to be **private** (intended for internal use); and the constructor **Employee** is declared to be **public.**

## Running a C++ program

It's time to put everything you've learned so far together into a complete compilable program. To compile a C++ program in the integrated environment, enter or load your text into the editor as usual. You can run C++ programs from the IDE in either of two ways. First, by default, any file with the .CPP extension will be compiled assuming C++ syntax, and any files with the .C extension will be compiled assuming C syntax. However, you can select the C++ Always button in the Source Options dialog box to have all files treated as C++ source files, regardless of extension.

To compile a C++ program with the command-line compiler, just give your file the extension .CPP. Or you can use the command-line option **–P**, in which case Turbo C++ will assume that the file has an extension of .CPP. If the file has a different extension, you must give the extension along with the file name. Life will be easier for you (and your next-of-kin) if you give all C++ programs a .CPP extension and all C programs a .C extension.

The program POINT.CPP defines the *Point* class and manipulates its data values:

*This code is available to load and run: POINT.CPP.*

```
/* POINT.CPP illustrates a simple Point class */

#include <iostream.h>          // needed for C++ I/O

class Point {                  // define Point class
   int X;                // X and Y are private by default
   int Y;
public:
   Point(int InitX, int InitY) {X = InitX; Y = InitY;}
   int GetX() {return X;}  // public member functions
   int GetY() {return Y;}
};

int main()
{
   int YourX, YourY;

   cout << "Set X coordinate: ";  // screen prompt
   cin >> YourX;                  // keyboard input to YourX

   cout << "Set Y coordinate: ";  // another prompt
   cin >> YourY;                  // key value for YourY

   Point YourPoint(YourX, YourY);  // declaration calls constructor

   cout << "X is " << YourPoint.GetX(); // call member function
   cout << '\n';                        // newline
   cout << "Y is " << YourPoint.GetY(); // call member function
   cout << '\n';
   return 0;
}
```

The class **Point** now contains a new member functions, **GetY**. This function works just like the **GetX** defined earlier, but accesses the private data member *Y* rather than *X*. Both are "short" functions and good candidates for the inline form of definition within the class body.

As with a macro using the **#define** directive, the code for an inline function is substituted directly into your file each time the function is used, thereby avoiding the function call overhead at the expense of code size. This is the classic "space versus time" dilemma found in many programming situations. As a general rule you should only use inline definitions for "short" functions, say one to three statements. Note that, unlike a macro, an inline function doesn't sacrifice the type checking that helps prevent errors in function calls. The number of arguments in a function is also relevant to your decision whether to "inline" or not, since the argument structure affects the function call overhead. The case for inlining is strongest when the total code for the function body is smaller than the code it takes to call the function out of line. You may need to try both methods and examine the assembly code output before deciding which approach is best for your needs.

Whether to inline a constructor or not can depend on whether base constructors are involved. A derived class constructor, especially where there are virtual functions (see page 156) in the hierarchy, can generate a lot of "hidden" code.

In the above example, the **Point** constructor has been defined as out-of-line, following the end of the class declaration. While you can put definitions in any order (and even put them elsewhere in the current file), it makes sense with smaller, single-file programs to put those definitions that aren't inline right after the class definition, in the order in which they were declared.

As your code gets larger, you'll probably have your class declarations in header files, and your class function definitions (implementation code) in separately compiled C++ source files. Inline function definitions, however, should always be in the header file.

This program also introduces the C++ **iostreams** library (note the statement `#include <iostream.h>` at the beginning of the program).

**cout** represents the *standard output stream* (by default, the screen). Data (variable values and strings, for example) are sent to it using the "put to" or insertion operator, **<<**.

**cin** represents the *standard input stream* (normally the keyboard). Values typed at the keyboard are stored in variables using the **>>** ("get from" or extraction) operator. The use of the shift operators, **>>** and **<<**, for stream I/O is a typical example of operator overloading in C++.

*The iostreams library is discussed in detail on page 184 and also in Chapter 3, "C++ streams," in the Programmer's Guide.*

The streams functions save you having to deal directly with the kinds of formatting details that **printf** and **scanf** require; they also allow I/O to be tailored to particular classes.

Once the X and Y values have been received from the keyboard, the **Point** object *YourPoint* is declared with the received values as arguments. Recall that this declaration automatically invokes the constructor for the **Point** class, which creates and initializes *YourPoint*.

Try running the program. The result should look like this:

```
Set X coordinate: 50
Set Y coordinate: 100
X is 50
Y is 100
```

## Inheritance

Classes don't usually exist in a vacuum. A program often has to work with several different but related data structures. For example, you might have a simple memory buffer in which you can store and from which you can retrieve data. Later, you may need to create more specialized buffers: A file buffer that holds data being moved to and from a file, and perhaps a buffer to hold data for a printer, and another to hold data coming from or going to a modem. These specialized buffers clearly have many characteristics in common, but each has some differences caused by the fact that disk files, printers, and modems involve devices that work differently.

The C++ solution to this "similar but different" situation is to allow classes to *inherit* characteristics and behavior from one or more *base* classes. This is an intuitive leap; inheritance is perhaps the single biggest difference between C++ and C. Classes that inherit from base classes are called *derived* classes. And a derived class may itself be the base class from which other classes are derived (recall the insect family tree).

### Rethinking the Point class

The fundamental unit of graphics is the single point on the screen (one pixel). So far we've devised several variants of a **Point** class that define a point by its *X* and *Y* locations, a constructor that creates and initializes a point's location, and other member functions that can return the point's current *X* and *Y* coordinates. Before you can draw anything, however, you have to distinguish between pixels that are "on" (drawn in some visible color) and pixels that are "off" (have the background color). Later, of course, you may want to define which of many colors a given point should have, and perhaps other attributes (such as blinking). Pretty soon you can end up with a complicated class that has many data members.

Let's rethink our strategy. What are the two fundamental *kinds* of information about points? One kind of information describes *where* the point is (location) and the other kind of information describes *how* the point is (the point's state of being: You can either see it, or you can't, and if you can see it, it is in some color). Of the two, the *location* is most fundamental: Without a location, you can't have a point at all.

Because all points must contain a location, you can make the class **Point** a *derived* class of a more fundamental *base* class, **Location**, which contains the information about *X* and *Y* coordinates. **Point** inherits everything that **Location** has, and adds whatever is new about **Point** to make **Point** what it must be.

These two related classes can be defined this way:

*This code is available as point.h.*

```
/* point.h--Example from Chapter 5 of Getting Started */

// point.h contains two classes:
// class Location describes screen locations in X and Y coordinates
// class Point describes whether a point is hidden or visible

enum Boolean {false, true};

class Location {
protected:          // allows derived class to access private data
   int X;
   int Y;

public:             // these functions can be accessed from outside
   Location(int InitX, int InitY);
   int GetX();
   int GetY();
};
class Point : public Location {      // derived from class Location
// public derivation means that X and Y are protected within Point

   protected:
   Boolean Visible;  // classes derived from Point will need access

public:
   Point(int InitX, int InitY);      // constructor
   void Show();
   void Hide();
   Boolean IsVisible();
   void MoveTo(int NewX, int NewY);
};
```

Here, **Location** is the base class, and **Point** is the derived class. The process can continue indefinitely: You can define other classes derived from **Location**, other classes derived from **Point**, yet more classes derived from **Point's** derived class, and so on. You can even have a class derived from more than one base class: This is called *multiple inheritance,* and will be discussed later. A large part of designing a C++ application lies in building this class hierarchy and expressing the family tree of the classes in the application.

## Inheritance and access control

Before we discuss the various member functions point.h, let's review the inheritance and access control mechanisms of C++.

The data members of the **Location** class are declared to be **protected**—recall that this means that member functions in both the **Location** class and the derived class **Point** will be able to access them, but the "public at large" won't be able to do so.

You declare a derived class as follows:

```
class D : access_modifier B { // default is private
   ...
}
```

or

```
struct D : access_modifier B { // default is public
   ...
}
```

**D** is the name of the derived class, *access_modifier* is optional (either **public** or **private**), and **B** is the name of the base class.

With **class**, the default *access_modifier* is **private**; with **struct**, the default is **public**. (Note that **unions** can be neither base nor derived classes.)

The *access_modifier* is used to modify the accessibility of inherited members, as shown in the following table:

Table 5.1
Class access

*In a derived class, access to the elements of its base class can be made* ***more*** *restrictive but never* ***less*** *restrictive.*

| Access in base class | Access modifier | Inherited access in base |
|---|---|---|
| public | public | public |
| private | public | not accessible |
| protected | public | protected |
| public | private | private |
| private | private | not accessible |
| protected | private | private |

When writing new classes that rely on existing classes, make sure you understand the relationship between base and derived classes. A vital part of this is understanding the access levels conferred by the specifiers **private, protected**, and **public**. Access rights must be passed on carefully (or withheld) from parents to children to grandchildren. C++ lets you do this without "exposing" your data to non-family and non-friends. The access level of a base class member, as viewed by the base class, need not be the same as its access level as viewed by its derived class. In other words, when members are inherited, you have some control over how their access levels are inherited.

*See Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide for more advanced technical details.*

A class can be derived privately or publicly from its base class. **private** derivation (the default for **class** type classes) converts **public** and **protected** members in the base class into **private** members of the derived class, while **private** members remain **private**. (Although **private** derivation is the default for classes, it is by no means the most commonly used method of derivation—so we have a rare situation where the default is not the norm).

A **public** derivation leaves the access level unchanged.

A derived class inherits all members of its base class, but can only use the **protected** and **public** members of its base class. **private** members of the base class are not directly available through the members of the derived class.

The particular definitions of **Location** and **Point** adopted here will allow us later on to derive further classes from **Point** for more complex graphics applications.

*Base class members that you want to use in a derived class must be either **protected** or **public**. **private** base class members can't be accessed except by their own member functions or through **friend** functions.*

If you use **public** derivation, **protected** members of the base class remain **protected** in the derived class, and thus won't be available from outside except to other publicly derived classes and friends. It's a good idea to always specify **public** or **private**, whatever the default, to avoid confusion. Good comments, too, will improve your source code legibility.

## Packaging classes into modules

Classes such as **Location** and **Point** can be packaged together for use in further program development. With its built-in data, member functions, and access control, a class is inherently modular. In developing a program, it often makes sense to put the declarations for each class or group of related classes in a separate header file, and the definitions for its non-inline member functions in a separate source file. (See Chapter 2, "Managing multi-file projects," in the *User's Guide* for details on how to use the Project

Manager to manage programs that consist of multiple source files.)

You can also combine several class object files into a library using TLIB. (See Chapter 5, "Utilities," in the *User's Guide* to learn how to create libraries.)

There are further advantages to modularizing classes: You can distribute your classes in object form to other programmers. The other programmers can derive new, specialized classes from the ones you made available, without needing access to your source code. Even though C++ version 2.0 is quite new, third-party class libraries are already appearing, and you can expect that your fellow C++ programmers will be offering many more goodies that you can use to get a head start in your programming projects.

We can now develop a separately compiled "module" containing the **Location** and **Point** classes. First, the declarations for the two classes (including their member functions) as listed on page 142 are put in the file point.h (on your distribution diskettes).

Note again how the class **Point** is derived from the class **Location**:

```
class Point : public Location { ...
```

The keyword **public** is needed before **Location** to ensure that the member functions of the derived class, **Point**, can access the protected members *X* and *Y* in the base class, **Location**. In addition to the *X* and *Y* location members, **Point** inherits the member functions **GetX** and **GetY** from **Location**. The class **Point** also adds the **protected** data member *Visible* (of the enumerated type *Boolean*), and five public member functions, including the constructor **Point::Point**. Note again that we have used **protected** rather than **private** access for certain elements so that point.h can be used in later examples that have further classes derived from **Location** and **Point**.

The file POINT2.CPP contains the definitions for all of the member functions of these two classes:

*This code is available as POINT2.CPP.*

```
/* POINT2.CPP--Example from Chapter 5 of Getting Started */

// POINT2.CPP contains the definitions for the Point and Location
// classes that are declared in the file point.h

#include "point.h"
#include <graphics.h>

// member functions for the Location class
Location::Location(int InitX, int InitY) {
   X = InitX;
   Y = InitY;
};

int Location::GetX(void) {
   return X;
};

int Location::GetY(void) {
   return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Point::Point(int InitX, int InitY) : Location(InitX,InitY) {
   Visible = false;                 // make invisible by default
};

void Point::Show(void) {
   Visible = true;
   putpixel(X, Y, getcolor());      // uses default color
};

void Point::Hide(void) {
   Visible = false;
   putpixel(X, Y, getbkcolor()); // uses background color to erase
};

Boolean Point::IsVisible(void) {
   return Visible;
};

void Point::MoveTo(int NewX, int NewY) {
   Hide();         // make current point invisible
   X = NewX;       // change X and Y coordinates to new location
   Y = NewY;
   Show();         // show point at new location
};
```

*A base constructor is invoked before the body of the derived class constructor.*

This example introduces the important concept of base-class constructors. When a **Point** object is defined, we want to make use of the fact that its base class, **Location**, already has its own user-defined constructor. The definition of the constructor **Point::Point** begins with a colon and a reference to the base constructor *Location(InitX,InitY)*. This specifies that the **Point** constructor will first call the **Location** constructor with the arguments *InitX* and *InitY*, thereby creating and initializing data members *X* and *Y*. Then the **Point** constructor body is invoked, creating and initializing the data member *Visible*. By explicitly specifying a base constructor, we have saved ourselves some coding (in larger examples, of course, the savings may be more significant).

In fact, derived-class constructors *always* call a constructor of the base class first to ensure that inherited data members are correctly created and initialized. If the base class is itself derived, the process of calling base constructors continues down the hierarchy. If you don't define a constructor for a particular class **X,** C++ will generate a default constructor of the form **X::X()**; that is, a constructor with no arguments.

If the derived-class constructor does not explicitly invoke one of its base-class constructors, or if you have not defined a base-class constructor, the default base class constructor (with no arguments) will be invoked. (There's more on base class constructors in Chapter 1, "The Turbo C++ language standard," in the *Programmer's Guide.*)

Notice that the reference to the base class constructor, *Location(InitX,InitY)* appears in the definition, not the declaration, of the derived class constructor.

Here's a main program (available on your distribution diskettes as PIXEL.CPP) that demonstrates the capabilities of the **Point** and **Location** classes.

*You'll need to compile and link POINT2.CPP, PIXEL.CPP, and GRAPHICS.LIB, using the PIXEL.PRJ project file supplied on your distribution diskettes. (Read Chapter 2, "Managing multi-file projects," in the User's Guide if you don't know how to use project files.)*

```
/* PIXEL.CPP--Example from Chapter 5 of Getting Started */

// PIXEL.CPP demonstrates the Point and Location classes
// compile with POINT2.CPP and link with GRAPHICS.LIB

#include <graphics.h>   // declarations for graphics library
#include <conio.h>      // for getch() function
#include "point.h"      // declarations for Point and Location
classes

int main()
{
   // initialize the graphics system
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "c:..\\bgi");

   // move a point across the screen
   Point APoint(100, 50);   // Initial X, Y at 100, 50
   APoint.Show();           // APoint turns itself on
   getch();                 // Wait for keypress
   APoint.MoveTo(300, 150); // APoint moves to 300,150
   getch();                 // Wait for keypress
   APoint.Hide();           // APoint turns itself off
   getch();                 // Wait for keypress
   closegraph();            // Restore original screen
   return 0;
}
```

## Extending classes

One of the beauties of classes is the way that new objects can be accommodated and given appropriate functionality. The next example takes the already defined **Location** and **Point** classes and derives a new class, **Circle**, along with functions to show, hide, expand, move, and contract circles.

*This code is on your disks: CIRCLE.CPP.*

```
/* CIRCLE.CPP--Example from Chapter 5 of Getting Started */

// CIRCLE.CPP   A Circle class derived from Point

#include <graphics.h>    // graphics library declarations
#include "point.h"       // Location and Point class declarations
#include <conio.h>       // for getch() function

// link with point2.obj and graphics.lib

class Circle : Point {   // derived privately from class Point
                                  // and ultimately from
class Location
   int Radius;           // private by default

public:
   Circle(int InitX, int InitY, int InitRadius);
   void Show(void);
   void Hide(void);
   void Expand(int ExpandBy);
   void MoveTo(int NewX, int NewY);
   void Contract(int ContractBy);
};

Circle::Circle(int InitX, int InitY, int InitRadius) :
Point(InitX,InitY)
{
   Radius = InitRadius;
};

void Circle::Show(void)
{
   Visible = true;
   circle(X, Y, Radius);      // draw the circle
}

void Circle::Hide(void)
{
   unsigned int TempColor;    // to save current color
   TempColor = getcolor();    // set to current color
   setcolor(getbkcolor());    // set drawing color to background
   Visible = false;
   circle(X, Y, Radius);      // draw in background color to erase
   setcolor(TempColor);       // set color back to current color
};

void Circle::Expand(int ExpandBy)
{
   Hide();                     // erase old circle
   Radius += ExpandBy;         // expand radius
   if (Radius < 0)             // avoid negative radius
      Radius = 0;
   Show();                     // draw new circle
};

void Circle::Contract(int ContractBy)
{
   Expand(-ContractBy);       // redraws with (Radius - ContractBy)
};

void Circle::MoveTo(int NewX, int NewY)
{
   Hide();                    // erase old circle
   X = NewX;                  // set new location
   Y = NewY;
   Show();                    // draw in new location
};

main()                        // test the functions
{
   // initialize the graphics system
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "c:..\\bgi");

   Circle MyCircle(100, 200, 50);   // declare a circle object
   MyCircle.Show();                 // show it
   getch();                         // wait for keypress
   MyCircle.MoveTo(200, 250);       // move the circle (tests hide
                                    // and show also)
   getch();
   MyCircle.Expand(50);             // make it bigger
   getch();
   MyCircle.Contract(75);           // make it smaller
   getch();
   closegraph();
   return 0;
}
```

To see how this works for the **Circle** class, you need to examine the member functions in the listing CIRCLE.CPP and refresh yourself on the class declarations in point.h.

Note first that the member functions of **Circle** need to access various data members in the classes **Circle, Point,** and **Location.**

Consider **Circle::Expand**. It needs access to **int** *Radius*. No problem. *Radius* is defined as **private** (by default) in **Circle** itself. So, *Radius* is accessible to **Circle::Expand**—indeed, it is accessible *only* to member functions of **Circle**. (Later, you'll see that the **private** members of a class can also be accessed by functions that have been specially defined as **friends** of that class.)

Next, look at the member function **Circle::Hide**. This needs to access **Boolean** *Visible* from its base class **Point**. Now *Visible* is protected in **Point**, and **Circle** is derived privately (by default) from **Point**. So, from the rules outlined above, *Visible* is **private** *within* **Circle**, and is accessible just like *Radius*. Note that if *Visible* had been defined as **private** in **Point**, it would have been inaccessible to the member functions of **Circle**. So, you might be tempted to make *Visible* **public**. However, this is overkill: *Visible* would become accessible to non-member functions. You might say that **protected** is **private** with a dash of **public** for derived classes: member functions of a derived class can access a **protected** member without exposing that member to public abuse.

Finally, consider **Circle::Show**. **Circle::Show** needs to access **Location**'s members *X* and *Y* in order to draw the circle. How is this achieved? **Circle** is not directly derived from **Location**, so the access rights are not immediately obvious. **Circle** derives from **Point** which derives from **Location**. Let's trace the access declarations.

1. Members *X* and *Y* are declared **protected** in **Location**.
2. **Point** specifies **public** derivation from **Location**, so **Point** also inherits the *X* and *Y* members as **protected**.
3. **Circle** is derived from **Point** using the default **private** derivation.
4. **Circle** therefore inherits *X* and *Y* as **private**. **Circle::Show** can access *X* and *Y*. Note that *X* and *Y* are still **protected** within **Location**.

Having digested this chain of access rights, you might want to consider the situation if a derived class of **Circle**, such as **PieChart** or **Arc**, was needed. Yes, you would need to change the derivation of **Circle** from **Point**—it would need to be a **public** derivation and *Radius* would need to become **protected**.

It should now be pretty easy to see what is going on in CIRCLE.CPP. A circle, in a sense, is a fat point: It has everything a point has (an *X,Y* location and a visible/invisible state) plus a radius. Class **Circle** appears to have only the single member *Radius*, but don't forget about all the members that **Circle** inherits by being a derived class of **Point**. **Circle** has *X*, *Y*, and *Visible* as well, even if you don't see them in the class definition for **Circle**.

Compile and link CIRCLE.CPP, POINT2.CPP, and GRAPHICS.LIB. The project file CIRCLE.PRJ on your distribution diskettes will help you do this. As you press a key, you should see a circle. Press a key again and the circle moves. Again, and the circle expands, and again and the circle contracts.

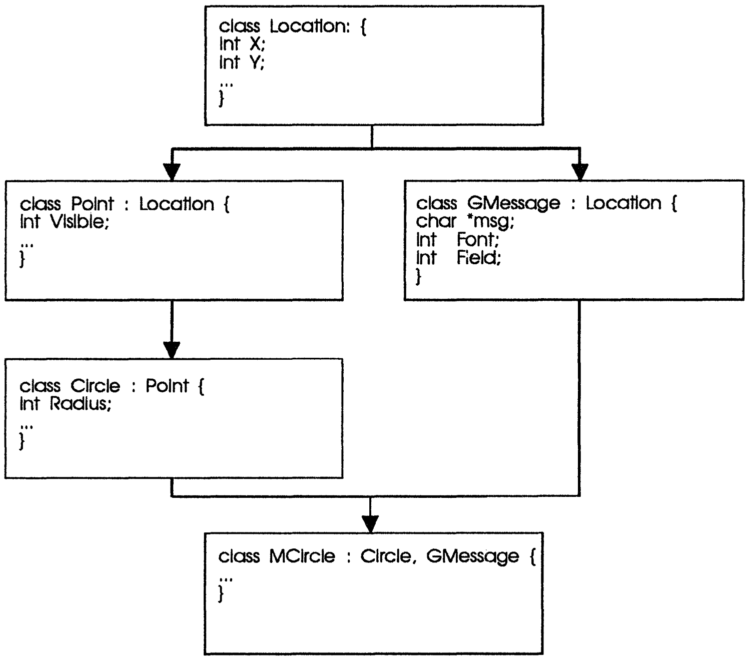
## Multiple inheritance

As we mentioned earlier, a class can inherit from more than one base class. This *multiple inheritance* mechanism was one of the main features added to C++ release 2.0. To see a practical example, the next program lets you display text inside a circle.

Your first thought might be to simply add a string data member to the *Circle* class and then add code to **Circle::Show** so that it displays the text with the circle drawn around it. But text and circles are really quite different things: When you think of text you think of fonts, character size, and possibly other attributes, none of which really has anything to do with circles. You could, of course, derive a new class directly from *Circle* and give it text capabilities. When dealing with fundamentally different functionalities, however, it is often better to create new "fundamental" base classes, and then derive specialized classes that combine the appropriate features. The next listing, MCIRCLE.CPP, illustrates this approach.

We'll define a new class called *GMessage* that displays a string on the screen starting at specified *X* and *Y* coordinates. This class will be *MCircle*'s other parent. *MCircle* will inherit **GMessage::Show** and use it to draw the text. The relationships of all of the classes involved is shown in the next figure.

Figure 5.3
Multiple inheritance



*This code is available on your disks: MCIRCLE.CPP. You need to run it using MCIRCLE.PRJ.*

```
/* MCIRCLE.CPP--Example for Chapter 5 of Getting Started */

// MCIRCLE.CPP       Illustrates multiple inheritance

#include <graphics.h> // Graphics library declarations
#include "point.h"    // Location and Point class declarations
#include <string.h>   // for string functions
#include <conio.h>    // for console I/O

// link with point2.obj and graphics.lib

// The class hierarchy:
// Location->Point->Circle
// (Circle and CMessage)->MCircle

class Circle : public Point {  // Derived from class Point and
ultimately
                                                // from class
Location
protected:
  int Radius;
public:
  Circle(int InitX, int InitY, int InitRadius);
  void Show(void);
};
```

```
class GMessage : public Location {
// display a message on graphics screen
   char *msg;              // message to be displayed
   int Font;               // BGI font to use
   int Field;              // size of field for text scaling

public:
   // Initialize message
   GMessage(int msgX, int msgY, int MsgFont, int FieldSize,
            char *text);
   void Show(void);        // show message
};


class MCircle : Circle, GMessage {  // inherits from both classes
public:
   MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
           char *msg);
   void Show(void);                 // show circle with message
};


// Member functions for Circle class

//Circle constructor
Circle::Circle(int InitX, int InitY, int InitRadius) :
   Point (InitX, InitY)       // initialize inherited members
//also invokes Location constructor
{
   Radius = InitRadius;
};

void Circle::Show(void)
{
   Visible = true;
   circle(X, Y, Radius); // draw the circle
}

// Member functions for GMessage class

//GMessage constructor
GMessage::GMessage(int msgX, int msgY, int MsgFont,
                       int FieldSize, char *text) :
                       Location(msgX, msgY)
//X and Y coordinates for centering message
{
   Font = MsgFont;    // standard fonts defined in graph.h
   Field = FieldSize; // width of area in which to fit text
   msg = text;        // point at message
};

void GMessage::Show(void)
{
   int size = Field / (8 * strlen(msg));     // 8 pixels per char.
   settextjustify(CENTER_TEXT, CENTER_TEXT); // centers in circle
   settextstyle(Font, HORIZ_DIR, size);      // if size > 1,
magnifies
   outtextxy(X, Y, msg);                     // display the text
}

//Member functions for MCircle class

//MCircle constructor
MCircle::MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
                        char *msg) : Circle (mcircX, mcircY,
mcircRadius),
                        GMessage(mcircX,mcircY,Font,2*mcircRadius,msg)
{
}

void MCircle::Show(void)
{
   Circle::Show();
   GMessage::Show();
}

main()      //draws some circles with text
{
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "c:..\\bgi");
   MCircle Small(250, 100, 25, SANS_SERIF_FONT, "You");
   Small.Show();
   MCircle Medium(250, 150, 100, TRIPLEX_FONT, "World");
   Medium.Show();
   MCircle Large(250, 250, 225, GOTHIC_FONT, "Universe");
   Large.Show();
   getch();
   closegraph();
   return 0;
}
```

As you read the listing, check the class declarations and note which data members and member functions are inherited by each class. You may also want to look at point.h again, since the *Location* and *Point* classes are defined there. Notice that both *MCircle* and *GMessage* have *Location* as their ultimate base class: *MCircle* by way of *Point* and *Circle*, and *GMessage* directly.

*The :: operator is used to specify a function from another scope rather than (by default) using the function of that name in the current scope.*

In the body of the definition of **MCircle::Show**, you will see the two function calls **Circle::Show();** and **GMessage::Show();**. This syntax shows another common use of **::** (the scope resolution operator). When you want to call an inherited function, such as **Show**, the compiler may need some help: which **Show** is required? Without the scope resolution "override," **Show()** would refer to the **Show()** in the current scope, namely **MCircle::Show()**. To call the **Show()** of another scope (assuming, of course, that you have access permission), you must supply the appropriate class name followed by **::** and the function name (with arguments, if any). What if there happened to be a *nonmember* function called **Show** that you wanted to call? You would use **::Show()** with no preceding class name.

*You'll find a more detailed account of how C++ handles scope in Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide.*

A member function of a given name in the derived class *overrides* the member function of the same name in the base class, but you can still get at the latter by using **::**. The scoping rules for C++ are slightly different from those for C.

Before leaving MCIRCLE.CPP, a brief word about the constructor for **MCircle.** You saw earlier how the **Point** constructor explicitly invoked its base constructor in **Location.** Since **MCircle** inherits from *both* **Circle** and **GMessage,** the **MCircle** constructor can conveniently initialize by calling *both* base constructors:

```
MCircle::MCircle
   (int mcircX, int mcircY, int mcircRadius, int font, char *msg) :
   Circle(mcircX, mcircY, mcircRadius),
   GMessage(mcircX, mcircY, 2*mcircRadius,msg) {
}
```

The constructor body is empty here because all the necessary work is accomplished in the *member initialization list* (after the : you enter a list of initializing expressions separated by commas. You met a simpler version of this syntax in the single base class constructors used in the **Point** and **Circle** class definitions). When the **MCircle** constructor is invoked (by declaring an **MCircle** object, for example), quite a spate of activity is triggered behind the scenes.

*See Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide for details on constructor calling sequences.*

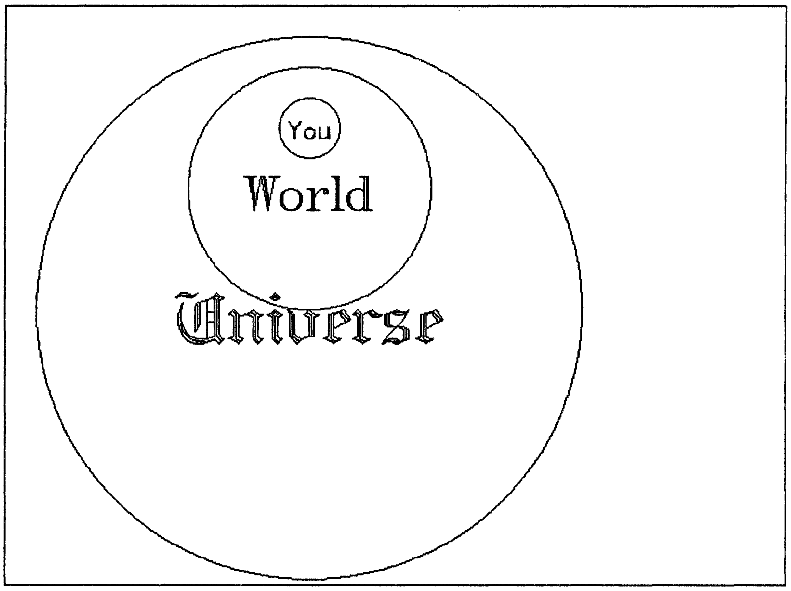
First, the **Circle** constructor is called. This constructor then calls the **Point** constructor, which in turn calls the **Location** constructor. Finally, the **GMessage** constructor is called, which calls the **Location** constructor for its own copy of its base class *X* and *Y*. The arguments given in the **MCircle** constructor are passed on to initialize the appropriate data members of the base classes.

When destructors are called (when an object goes out of scope, for example), the deallocation sequence is the reverse of that used during construction. (Virtual base class constructors and destructors have some sequencing quirks beyond the scope of this chapter).

In passing, recall the point made earlier: if you don't supply your own constructors or destructors, C++ will generate and invoke default versions behind the scenes.

Figure 5.4 shows the output of MCIRCLE:

Figure 5.4
Circles with messages



## Virtual functions

Each class type in our graphics hierarchy represents a different type of figure onscreen: a point or a circle. 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 classes to represent other figures such as lines, squares, arcs, and so on, you could write a member function for each that would display that object onscreen. In the new way of object-oriented thinking, you could say that all these graphic figure types had the ability to show themselves on the screen.

What is different for each object type is the *way* it must show itself onscreen. A point is drawn with a point-plotting routine that needs only an *X*,*Y* location and perhaps a color value. A circle needs a more complex graphics routine to display itself, taking into account not only *X* and *Y*, but a radius as well. Still further, an arc needs a start angle and an end angle, and a different drawing algorithm. The same situation, of course, applies to hiding, dragging, and other basic shape manipulations.

The ordinary member functions you have seen so far certainly allow us to define a **Show** function for each shape class. But they lack an essential ingredient. Graphics modules based on our existing classes and member functions would need source code changes and recompilations each time a new shape class was introduced with its own member function **Show**. The reason is that the C++ mechanisms revealed so far allow essentially only three ways to resolve the question: which **Show** is being referenced?:

1. There's the distinction by argument signature—**Show(int,char)** is not the same function as **Show(char*,float)**, for example.
2. There's the use of the scope resolution operator, whereby **Circle::Show** is distinguished from **Point::Show** and **::Show**.
3. There's the resolution by class object: *ACircle.Show* invokes **Circle::Show**, while *Apoint.Show* invokes **Point::Show**. Similarly with pointers to objects: *APoint_pointer->Show* invokes **Point::Show**.

All these function resolutions, so far, have been made at compile time—a mechanism which is referred to as *early* or *static* binding.

A typical graphics toolbox would provide the user with class definitions in .H source files together with the precompiled .OBJ or .LIB code for the member functions. With the early binding restrictions, the user cannot easily add new class shapes, and even the developer faces extra chores in extending the package. C++ offers a flexible mechanism to solve these problems: *late* (or *dynamic*) binding by means of special member functions called *virtual* functions.

The key concept is that virtual function calls are resolved at run time (hence the term, late binding). In practical terms, it means that the decision as to which **Show** function is called can be deferred until the object type involved is known during execution. A virtual function **Show**, "hidden" in a class **B** in the precompiled toolbox library, is not *bound* to the objects of **B** in the way that ordinary member functions of **B** are. You are free to create a class **D** derived from **B** for your own favorite shape, and write appropriate functions (putting on your **Show**, as it were). You then compile and link your OBJ or LIB code to that of the toolbox. Calls made on **Show**, whether from existing member functions of

**B** *or* from the new functions you have written for **D**, will automatically reference the correct **Show**. This resolution is made entirely on the object type involved in the call. Let's look at virtual functions in action. We have a potential candidate in the earlier code given for CIRCLE.CPP

## Virtual functions in action

Consider the member function **Circle::MoveTo** in CIRCLE.CPP:

```
void Circle::MoveTo(int NewX, int NewY)
{
   Boolean vis = Visible;
   if (vis) Hide();    // hide only if visible
   X = NewX; Y = NewY; // set new location
   if (vis) Show();    // draw at new location if previously
                       // visible
}
```

Notice how similar this definition is to **Point::MoveTo** found in the **Circle**'s base class **Point**. In fact, the return value, function name, number and types of formal arguments (known as the function *signature*), and even the function body itself, all appear to be identical! If C++ encounters two function calls using the same function name but differing in signatures, we have already seen that the C++ compiler is smart enough to resolve the potential ambiguities caused by function-name *overloading*. (Recall that C, unlike C++, demands unique function names.) In C++, member functions with different signatures are really different functions, even if they share the same name.

But, our two **MoveTo**s do not, at first sight, offer any distinguishing clues to the compiler—so will it know which one you intended to call? The answer, as you've seen, with ordinary member functions is that the compiler determines the target function from the class type of the object involved in the call.

So, why not let **Circle** inherit **Point**'s **MoveTo**, just as **Circle** inherits **Point**'s **GetX** and **GetY** (via **Location**)? The reason, of course, is that the **Hide** and **Show** called in **Circle::MoveTo** are not the same **Hide** and **Show** called in **Point::MoveTo**. Only the names and signatures are the same. Inheriting **MoveTo** from **Point** would lead to the wrong **Hide** and **Show** being called when trying to move a circle. Why? Because **Point**'s versions of these two functions would be bound to **Point**'s (and hence also to **Circle**'s) **MoveTo** at compile time (early binding). As you may have guessed already, the answer is to declare **Hide** and **Show** as virtual functions. This will delay the binding so that the correct versions **Hide** and **Show** can be invoked when **MoveTo** is actually called to move a point or a circle (or whatever).

Note again that if we wanted to precompile our class definitions and member functions for **Location, Point**, and **Circle** in a neat standalone library (with the implementation source locked up with our other trade secrets), we certainly could not know in advance the objects that **MoveTo** may be asked to move. Virtual functions not only provide this technical advantage; they also provide a conceptual gain that lies at the heart of OOP. We can concentrate on developing reusable classes and methods with less anxiety about name clashes.

While it is true that add-on library extensions are available for most languages, the use of virtual functions and multiple inheritance in C++ makes extensibility more natural. You inherit everything that all your base classes have, and then you add the new capabilities you need to make new objects work in familiar ways. The classes you define and their versions of the virtual functions become a true extension of an orderly hierarchy of capabilities. Because this is part of the language design rather than an afterthought, there is very little penalty in performance.

Having sold you on the merits of virtual functions, let's see how you can implement them, and some of the rules you have to follow.

## Defining virtual functions

The syntax is straightforward: add the qualifier **virtual** in the member function's first declaration:

```
virtual void Show();
virtual void Hide();
```

Important! Only member functions can be declared as **virtual**. Once a function is declared **virtual**, it must not be redeclared in any derived class with the *same* formal argument signature but with a *different* return type. If you redeclare **Show** with the same formal argument signature and same return type, the new **Show** automatically becomes virtual, whether you use the **virtual** qualifier or not. This new, virtual **Show** is said to override the **Show** in its base class.

You are free to redeclare **Show** with a different formal argument signature (whether you change the return type or not)—but the virtual mechanism is inoperable for this version of **Show**. Beginners should avoid rash overloading—there are situations where a non-virtual function can hide a virtual function declared in its base.

The particular **Show** called will depend only on the class of the object for which **Show** is invoked, even if the call is invoked via a pointer (or reference) to the base class. For example,

```
Circle ACircle;
Point* APoint_pointer = &ACircle; // pointer to Circle assigned to
                                  // pointer to base class, Point
APoint_pointer->Show();           // calls Circle::Show!
```

vpoint.h and VCIRC.CPP (available on your distribution diskettes) are versions of point.h and CIRCLE.CPP with **Show** and **Hide** made virtual. Compile VCIRC.CPP with POINT2.CPP using VCIRC.PRJ. It will run exactly like CIRCLE.CPP. We don't list the virtual versions in full here since the differences can be summed up simply as follows:

- In vpoint.h, **Point**'s **Show** and **Hide** have been declared with the keyword **virtual**. The **Show** and **Hide** in the VCIRC's derived class **Circle** have the same argument signature and return values as the base versions in **Point**; this implies that they are also virtual, even though the keyword **virtual** is not used in their declarations.
- In VCIRC.CPP, **Circle** no longer has its own **MoveTo** member function.
- We now derive **Circle** publicly from **Point** to allow access to **MoveTo**

To recap the significance of these changes:

**Circle** objects can now safely call the **MoveTo** inherited from **Point**. The **Show** and **Hide** called by **MoveTo** will be bound at run time to **Circle**'s own **Show** and **Hide**. Any **Point** objects calling **MoveTo** will invoke the **Point** versions.

## Developing a complete graphics module

As a more complete and realistic example of virtual functions, let's create a module that defines some shape classes and a generalized means of dragging them around the screen. This module, figures.h and FIGURES.CPP (on your distribution diskettes), is a simple implementation of the graphics toolbox discussed earlier.

A major goal in designing the FIGURES module is to allow users of the module to extend the classes defined in the module—and still make use of all the module's features. It is an interesting challenge to create some means of dragging an arbitrary graphics figure around the screen in response to user input.

As a first approach, we might consider a function that takes an object as an argument, and then drags that object around the screen:

```
void Drag(Point& AnyFigure, int DragBy)
{
  int DeltaX,DeltaY;
  int FigureX,FigureY;
  AnyFigure.Show();          // Display figure to be dragged
  FigureX = AnyFigure.GetX(); // Get the initial X,Y of figure
  FigureY = AnyFigure.GetY();

  // This is the drag loop
  while (GetDelta(DeltaX, DeltaY))
  {
    // Apply delta to figure X,Y
    FigureX = FigureX + (DeltaX * DragBy);
    FigureY = FigureY + (DeltaY * DragBy);
    // And tell the figure to move
    AnyFigure.MoveTo(FigureX, FigureY);
  };
};
```

## Reference types

Notice that *AnyFigure* is declared to be of type **Point&**. This means "a reference to an object of type **Point**" and is a new feature of C++. As you know, C ordinarily passes arguments by value, not by reference. In C, if you want to act directly on a variable being passed to a function, you have to pass a pointer to the variable, which can lead to awkward syntax, since you have to remember to dereference the pointer. C++ lets you pass and modify the actual variable by using a reference. To declare a reference, simply follow the data type with an ampersand (**&**) in the variable declaration.

**Drag** calls an auxiliary function not shown here, **GetDelta**, that obtains some sort of change in *X* and *Y* from the user. It could be from the keyboard, or from a mouse, or a joystick. (For simplicity's sake, our example obtains input from the arrow keys on the keyboard.)

An important point to notice about **Drag** is that any object of type **Point**, or any type derived from **Point**, can be passed in the *AnyFigure* reference argument. Objects of **Point** or **Circle** type, or any type defined in the future that inherits from **Point** or **Circle**, can be passed without complication in *AnyFigure*.

Adding a new member function to an existing class hierarchy involves a little thought. How far up the hierarchy should the member function be placed? Think about the utility provided by the function and decide how broadly applicable that utility is. Dragging a figure involves changing the location of the figure in response to input from the user. In terms of inheritability, it sits right beside **MoveTo**—any object to which **MoveTo** is appropriate should also inherit **Drag**. Therefore **Drag** should be a member of **Point**, so that all of **Point**'s derived types can share it.

Having resolved the place of **Drag** in the hierarchy, we can take a closer look at its definition. As a member function of the base class **Point**, there is no need for the explicit reference to the *Point& AnyFigure* argument. We can rewrite **Drag** so that the functions it calls, such as **GetX**, **Show**, **MoveTo**, and **Hide**, will correctly reference the versions appropriate to the type of the object being dragged. As we saw earlier, the functions **Show** and **Hide** that require special shape-related code can be made virtual. We can then redefine them for any future classes without disturbing the FIGURES module. This also takes care of **MoveTo**, since **MoveTo** calls the correct **Show** and **Hide** (you'll recall that that was our original motivation for making **Show** and **Hide** virtual). **GetX** and **GetY** present no problem: as ordinary member functions inherited from **Point** via **Location**, they simply return the *X* and *Y* data members of the calling object of any derived class, present or future. Remember, though, that *X* and *Y* are protected in **Location**, so we must use public derivation as shown.

The next design decision is whether to make **Drag** virtual. The litmus test for making any function virtual is whether its functionality is expected to change somewhere down the hierarchy. There is no golden rule here, but later on we'll discuss the various tradeoffs: extensibility versus performance overhead (virtual functions require slightly more memory and a few more memory-access cycles). We have taken the view that some future class in, say, a CAD (Computer Aided Design) application might conceivably need a special dragging action. Perhaps dragging an isometric drawing will require some scaling actions, and so on. In our new **Point** class definition in figures.h, we have therefore made **Drag** virtual.

*Remember to recompile everything that uses this header file.*

```
class Point : public Location {
protected:
   Boolean Visible;
public:
   Point(int InitX, int InitY);
   virtual void Show();       // Show and Hide are virtual
   virtual void Hide();
   Boolean IsVisible() {return Visible;}
   void MoveTo(int NewX, int NewY);
   virtual void Drag(int DragBy);
};
```

Here is the header file figures.h containing the class declarations for the FIGURES module. This is the only part of the package that needs to be distributed in source code form:

*This code is on your disks: figures.h.*

```
// figures.h contains three classes.
//
//  Class Location describes screen locations in X and Y
//  coordinates.
//
//  Class Point describes whether a point is hidden or visible.
//
//  Class Circle describes the radius of a circle around a point.
//
// To use this module, put #include <figures.h> in your main
// source file and compile the source file FIGURES.CPP together
// with your main source file.

enum Boolean {false, true};

class Location {
protected:
   int X;
   int Y;
public:
   Location(int InitX, int InitY) {X = InitX; Y = InitY;}
   int GetX() {return X;}
   int GetY() {return Y;}
};

class Point : public Location {
protected:
   Boolean Visible;
public:
   Point(int InitX, int InitY);
   virtual void Show();       // Show and Hide are virtual
   virtual void Hide();
   virtual void Drag(int DragBy); // new virtual drag function
   Boolean IsVisible() {return Visible;}
   void MoveTo(int NewX, int NewY);
};

class Circle : public Point {  // Derived from class Point and
                               // ultimately from class Location
protected:
   int Radius;
public:
   Circle(int InitX, int InitY, int InitRadius);
   void Show();
   void Hide();
   void Expand(int ExpandBy);
   void Contract(int ContractBy);
};

// prototype of general-purpose, non-member function
// defined in FIGURES.CPP

Boolean GetDelta(int& DeltaX, int& DeltaY);
```

Here is the file FIGURES.CPP containing the member function definitions. This is what would be distributed in object or library form commercially. Note that we have defined the **Circle** constructor outside the class since it invokes base constructors. You may wish to experiment by making it an inline function (see the discussion on page 175). The nonmember function **GetDelta** will repay some study if you are new to C. Note the use of reference arguments, which is a C++ touch; the rest of the code is traditional.

*This code is on your disks: FIGURES.CPP. You should compile this code and link it to GRAPHICS.LIB to get FIGURES.OBJ. You'll need FIGURES.OBJ for the next exercise.*

```
// FIGURES.CPP: This file contains the definitions for the Point
// class (declared in figures.h). Member functions for the
// Location class appear as inline functions in figures.h.

#include "figures.h"
#include <graphics.h>
#include <conio.h>

// member functions for the Point class

//constructor
Point::Point(int InitX, int InitY) : Location (InitX, InitY)
{
   Visible = false;    // make invisible by default
}

void Point::Show()
{
   Visible = true;
   putpixel(X, Y, getcolor()); // uses default color
}

void Point::Hide()
{
   Visible = false;
   putpixel(X, Y, getbkcolor()); // uses background color to erase
}

void Point::MoveTo(int NewX, int NewY)
{
   Hide();          // make current point invisible
   X = NewX;        // change X and Y coordinates to new location
   Y = NewY;
   Show();          // show point at new location
}

// a general-purpose function for getting keyboard
// cursor movement keys (not a member function)

Boolean GetDelta(int& DeltaX, int& DeltaY)
{
   char KeyChar;
   Boolean Quit;
   DeltaX = 0;
   DeltaY = 0;

   do
{
      KeyChar = getch();     // read the keystroke
      if (KeyChar == 13)     // carriage return
         return(false);
      if (KeyChar == 0)      // an extended keycode
         {
          Quit = true;       // assume it is usable
          KeyChar = getch(); // get rest of keycode
              switch (KeyChar) {
                 case 72: DeltaY = -1; break; // down arrow
                 case 80: DeltaY =  1; break; // up arrow
                 case 75: DeltaX = -1; break; // left arrow
                 case 77: DeltaX =  1; break; // right arrow
                 default: Quit = false; // bad key
                 };
         };
   } while (!Quit);
   return(true);
}

void Point::Drag(int DragBy)
{
   int DeltaX, DeltaY;
   int FigureX, FigureY;

   Show(); // display figure to be dragged
     FigureX = GetX(); // get initial position of figure
   FigureY = GetY();

   // This is the drag loop
   while (GetDelta(DeltaX, DeltaY))
{
      // Apply delta to figure at X, Y
      FigureX += (DeltaX * DragBy);
      FigureY += (DeltaY * DragBy);
      MoveTo(FigureX, FigureY); // tell figure to move
      };
}
// Member functions for the Circle class

//constructor
Circle::Circle(int InitX, int InitY, int InitRadius) : Point (InitX,
InitY)
{
   Radius = InitRadius;
}

void Circle::Show()
{
   Visible = true;
   circle(X, Y, Radius);     // draw the circle
}

void Circle::Hide()
{
   unsigned int TempColor;   // to save current color
   TempColor = getcolor();   // set to current color
   setcolor(getbkcolor());   // set drawing color to background
   Visible = false;
   circle(X, Y, Radius);     // draw in background color to
   setcolor(TempColor);      // set color back to current color
}

void Circle::Expand(int ExpandBy)
{
   Hide();                          // erase old circle
   Radius += ExpandBy;              // expand radius
   if (Radius < 0)                  // avoid negative radius
      Radius = 0;
   Show();                          // draw new circle
}

void Circle::Contract(int ContractBy)
{
   Expand(-ContractBy);      // redraws with (Radius-ContractBy)
```

```
}
```

We are now ready to test FIGURES by exposing it to a new shape class called **Arc** that is defined in FIGDEMO.CPP. **Arc** is (naturally) derived publicly from **Circle**. Recall that **Drag** is about to drag a shape it has never seen before!

*This code is on your disks as FIGDEMO.CPP. You need to compile it and link it to FIGURES.OBJ.*

```
// FIGDEMO.CPP -- Exercise for Chapter 5

// demonstrates the Figures toolbox by extending it with
// a new type Arc.

// Link with FIGURES.OBJ and GRAPHICS.LIB

#include "figures.h"
#include <graphics.h>
#include <conio.h>

class Arc : public Circle {
   int StartAngle;
   int EndAngle;
public:
// constructor
   Arc(int InitX, int InitY, int InitRadius, int InitStartAngle, int
       InitEndAngle) : Circle (InitX, InitY, InitRadius) {
       StartAngle = InitStartAngle; EndAngle = InitEndAngle;}
   void Show();  // these functions are virtual in Point
   void Hide();
};

// Member functions for Arc

void Arc::Show()
{
   Visible = true;
   arc(X, Y, StartAngle, EndAngle, Radius);
}

void Arc::Hide()
{
   int TempColor;
   TempColor = getcolor();
   setcolor (getbkcolor());
   Visible = false;
   // draw arc in background color to hide it
   arc(X, Y, StartAngle, EndAngle, Radius);
   setcolor(TempColor);
}

int main()   // test the new Arc class
{
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "c:..\\bgi");
```

```
      Circle ACircle(151, 82, 50);
      Arc AnArc(151, 82, 25, 0, 190);

      // you first drag an arc using arrow keys (5 pixels per key)
      // press Enter when tired of this!
      // Now drag a circle (10 pixels per arrow key)
      // Press Enter to end FIGDEMO.

      AnArc.Drag(5);   // drag increment is 5 pixels
      AnArc.Hide();
      ACircle.Drag(10); // now each drag is 10 pixels
      closegraph();
      return 0;
   }
```

### Ordinary or virtual member functions?

In general, because calling a non-virtual member function is a little faster than calling a virtual one, we recommend that you use ordinary member functions when extensibility is not a consideration, but performance is. Use virtual functions otherwise.

To recap our earlier discussion, let's say you are declaring a class named **Base**, and within **Base** you are declaring a member function named **Action**. How do you decide whether **Action** should be virtual or ordinary? Here's the rule of thumb: Make **Action** virtual if there is a possibility that some future class derived from **Base** will override **Action**, and you want that future code to be accessible to **Base**. Make **Action** ordinary if it is evident that for derived types, **Action** will perform the same steps (even if this involves invoking other, virtual, functions); or the derived types will not make use of **Action**.

## Dynamic objects

All the examples shown so far, except for the message array allocation in MCIRCLE.CPP, have had static or automatic objects of class types that were declared as usual with their memory being allocated by the compiler at compile time. In this section we look at objects that are created at run time, with their memory allocated from the system's *free memory store*. The creation of dynamic objects is an important technique for many programming applications where the amount of data to be stored in memory cannot be known before the program is run. An example is a free-form database program that holds data records of various sizes in memory for rapid access.

C++ can use the dynamic memory allocation functions of C such as **malloc**. However, C++ includes some powerful extensions that make dynamic allocation and deallocation of objects easier and more reliable. More importantly, it ensures that constructors and destructors are called. For example,

*To allocate an object from free store, declare a pointer to the object's type and assign the result of the expression* `new object_type` *to the pointer. You can now use the pointer to refer to the newly created object.*

```
Circle *ACircle = new Circle(151,82,50);
```

Here *ACircle*, a pointer to type **Circle**, is given the address of a block of memory large enough to hold one object of type **Circle**. In other words, *ACircle* now points to a **Circle** object allocated from free store. A *Circle* constructor is then called to initialize the object according to the arguments supplied.

If you are allocating an array rather than a standard-length data type, use the optional syntax

**new** *object* [*size*]

For example, to dynamically allocate an array of 50 integers called *counts*, use

```
counts = new int [50];
```

If you wanted to create a dynamic **Point** class object, you might do it like this:

*You can find this on your disks: DYNPOINT.CPP. Or use DYNPOINT.PRJ.*

```
// DPOINT.CPP -- exercise in Chapter 5, Getting Started

#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "figures.h"

int main()
{
// Assign pointer to dynamically allocated object; call constructor
Point *APoint = new Point(50, 100);

// initialize the graphics system
int graphdriver = DETECT, graphmode;
initgraph(&graphdriver, &graphmode, "..\\bgi");

// Demonstrate the new object
APoint->Show();
cout << "Note pixel at (50,100). Now, hit any key...";
getch();
delete APoint;
closegraph();
return(0);
}
```

## Destructors and delete

Just as you can define a constructor that will be called whenever a new object of a class is created, you can define a *destructor* that will be called when it is time to destroy an object, that is to say, clear its value and deallocate its memory.

Space for static objects is allocated by the compiler; the constructor is called before **main** and the destructor is called after **main**. In the case of **auto** objects, deallocation occurs when the declaration goes out of scope (when the enclosing block terminates). Any destructor you define is called at the time the static or auto objects is destroyed. (If you haven't defined a destructor, C++ uses an implicit, or built-in one.)

If you create a dynamic object using the **new** operator, however, you are responsible for deallocating it, since C++ has no way of "knowing" when the object is no longer needed. You use the **delete** operator to deallocate the memory. Any destructor you have defined is called when **delete** is executed.

The **delete** operator has the syntax

*You can find more on destructor syntax in Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide.*

**delete** *pointer*;

where *pointer* is the pointer that was used with **new** to allocate the memory.

You have seen that a constructor for the class **X** is identified by having the same name, viz **X::X()**. The name of a destructor for class **X** is **X::~X()**. In addition to deallocating memory, destructors can also perform other appropriate actions, such as writing member field data to disk, closing files, and so on.

## An example of dynamic object allocation

The next example program provides some practice in the use of objects allocated dynamically from free store, including the use of destructors for object deallocation. The program shows how a linked list of graphics objects might be created in memory and cleaned up using delete calls when the objects 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 **Point** contains no such pointer. The easy way out would be to add a pointer to **Point**, and in doing so ensure that all **Point's** derived types also inherit the pointer. However, adding anything to **Point** requires that you have the source code for **Point**, and as noted earlier, one advantage of C++ is the ability to extend existing objects without necessarily being able to recompile them. So for this example we'll pretend that we don't have the source code to **Point** and show how you can extend the graphics tool kit anyway.

*See the next listing for the declarations of List and Node.*

One of the many solutions that requires no changes to **Point** is to create a new class not derived from **Point**. Type **List** is a very simple class whose purpose is to head up a list of **Point** objects. Because **Point** contains no pointer to the next object in the list, a simple **struct, Node,** provides that service. **Node** is even simpler than **List**, in that it has no member functions and contains no data except a pointer to type **Point** and a pointer to the next node in the list.

**List** has a member function that allows it to add new figures to its linked list of **Node** records by inserting a new **Node** object immediately after itself, as a referent to its Nodes pointer member. The **Add** member function takes a pointer to a **Point** object, rather than a **Point** object itself. Remember that rules for the class hierarchy in C++ allows pointers to any type publicly derived from **Point** to be passed in the *Item* argument to **List::Add**.

Program ListDemo declares a static variable, *AList*, of type **List**, and builds a linked list with three nodes. Each node points to a different graphics figure that is either a **Point** or one of its derived classes. The number of bytes of free storage 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 three **Node** records and the three **Point** objects, is cleaned up and removed from memory, thanks to the destructor for the **List** class called automatically for its object *AList*.

*This code is on your disks as LISTDEMO.CPP.*

```
/* LISTDEMO.CPP--Example from Chapter 5 of Getting Started */

// LISTDEMO.CPP          Demonstrates dynamic objects

// Link with FIGURES.OBJ and GRAPHICS.LIB

#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
```

```
#include <graphics.h>
#include "figures.h"

class Arc : public Circle {
   int StartAngle, EndAngle;
public:
   // constructor
   Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
       int InitEndAngle);
   // virtual functions
   void Show();
   void Hide();
};

struct Node {      // the list item
   Point *Item;    // can be Point or any class derived from Point
   Node  *Next;    // point to next Node object
};

class List {       // the list of objects pointed to by nodes
   Node *Nodes;    // points to a node
public:
   // constructor
   List();
   // destructor
   ~List();
   // add an item to list
   void Add(Point *NewItem);
   // list the items
   void Report();
};

// definitions for standalone functions

void OutTextLn(char *TheText)
{
   outtext(TheText);
   moveto(0, gety() + 12);   // move to equivalent of next line
}

void MemStatus(char *StatusMessage)
{
   unsigned long MemLeft;  // to match type returned by
            // coreleft()
   char CharString[12];    // temp string to send to outtext()
   outtext(StatusMessage);
   MemLeft = long (coreleft());

   // convert result to string with ltoa then copy into
   // temporary string
   ltoa(MemLeft, CharString, 10);
   OutTextLn(CharString);
```

```
}

// member functions for Arc class

Arc::Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
         int InitEndAngle) : Circle (InitX, InitY,InitRadius)
                             // calls Circle
                             // constructor
{
   StartAngle = InitStartAngle;
   EndAngle = InitEndAngle;
}

void Arc::Show()
{
   Visible = true;
   arc(X, Y, StartAngle, EndAngle, Radius);
}

void Arc::Hide()
{
   unsigned TempColor;
   TempColor = getcolor();
   setcolor(getbkcolor());
   Visible = false;
   arc(X, Y, StartAngle, EndAngle, Radius);
   setcolor(TempColor);
}

// member functions for List class

List::List () {
   Node *N;
   N = new Node;
   N->Item = NULL;
   N->Next = NULL;
   Nodes = NULL;              // sets node pointer to "empty"
                              // because nothing in list yet
}

List::~List()                 // destructor
{
   while (Nodes != NULL) {    // until end of list
      Node *N = Nodes;        // get node pointed to
      delete(N->Item);        // delete item's memory
      Nodes = N->Next;        // point to next node
        delete N;             // delete pointer's memory
   };
}

void List::Add(Point *NewItem)
{
   Node *N;               // N is pointer to a node
   N = new Node;         // create a new node
   N->Item = NewItem;    // store pointer to object in node
   N->Next = Nodes;      // next item points to curent list pos
   Nodes = N;            // last item in list now points
                         // to this node
}

void List::Report()
{
   char TempString[12];
   Node *Current = Nodes;
   while (Current != NULL)
   {
      // get X value of item in current node and convert to string
      itoa(Current->Item->GetX(), TempString, 10);
      outtext("X = ");
      OutTextLn(TempString);
      // do the same thing for the Y value
      itoa(Current->Item->GetY(), TempString, 10);
      outtext("Y = ");
      OutTextLn(TempString);
      // point to the next node
      Current = Current->Next;
   };
}

void setlist(void);

// Main program
main()
{
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "c:..\\bgi");

   MemStatus("Free memory before list is allocated: ");
   setlist();
   MemStatus("Free memory after List destructor: ");
   getch();
   closegraph();
}

void setlist() {

   // declare a list (calls List constructor)
   List AList;

   // create and add several figures to the list
   Arc *Arc1 = new Arc(151, 82, 25, 200, 330);
   AList.Add(Arc1);
   MemStatus("Free memory after adding arc1: ");
   Circle *Circle1 = new Circle(200, 80, 40);
   AList.Add(Circle1);
```

```
    MemStatus("Free memory after adding circle1: ");
    Circle *Circle2 = new Circle(305, 136, 35);
    AList.Add(Circle2);
    MemStatus("Free memory after adding circle2: ");
    // traverse list and display X, Y of the list's figures
    AList.Report();
    // The 3 Alist nodes and the Arc and Circle objects will be
    // deallocated automatically by their destructors when they
    // go out of scope in main(). Arc and Circle use implicit
    // destructors in contrast to the explicit ~List destructor.
    // However, you could delete explicitly here if you wish:
    // delete Arc1; delete Circle1; delete Circle2;
    getch();   // wait for a keypress
    return;
}
```

Once you have mastered LISTDEMO.CPP, you might wish to develop a more satisfying solution based on the following idea: define a new class called **PointList** by multiple inheritance from classes **Point** and **List**.

## More flexibility in C++

Although it will take you some time to master the nuances of this new style of programming, you have now learned the essential elements of C++. There are a number of additional features that we touch on briefly here so that you will know what they are and how to use them.

*None of these features are essential to understanding C++, but they can add to its flexibility and power.*

- Inline functions outside class definitions
- Default function arguments
- Overloading functions and multiple constructors
- Friend functions—another way of providing access to a class
- Overloading operators to provide new meanings
- More about C++ I/O and the streams library

### Inline functions outside class definitions

You have already seen that you can include an *inline* definition of a member function within the class declaration as shown here with the **Point** class:

```
class Point: {   // define Point class
   int X;        // these are private by default
```

```
    int Y;
public:         // public member functions
    Point(int InitX, int InitY) {X = InitX, Y = InitY;}
    int GetX(void) {return X;}
    int GetY(void) {return Y;}
};
```

All three member functions of the **Point** class are defined inline, so no separate definition is necessary. For functions with only a line or so of code, this provides a more compact, easier to read description of the class.

*Remember that inline code is enclosed in braces.*

Functions can also be declared as *inline*. The only difference is that you have to start the function declaration with the keyword **inline**. For example, in LISTDEMO.CPP, there is an operation that simply moves the output location for text in graphics mode down one line (it is used in the function *OutTextLn*). If this function were to be used in many other places in the code, it would be more efficient to declare it as a separate inline function:

```
inline void graphLn() { moveto(0, gety() + 12); }
```

If you wish, you can format your inline definitions to look more like a regular function definition:

```
inline void graphLn()
{
   moveto(0, gety() + 12);
}
```

Another advantage to using the **inline** keyword is that you can avoid revealing your implementation code in the distributed header files.

## Functions with default arguments

*If you plan to use certain values often for a function, use those values as default arguments for the function. Default values must be specified the first time the function name is given.*

You can define functions that you can call with fewer arguments than defined. The arguments that you don't supply are given default values. If you are going to be using these default values most of the time, such an "abbreviated" call saves typing. You don't lose flexibility, because when you want to override the defaults, you simply specify the values you want.

For example, the following version of the constructor for the *Circle* class gives a default circle of radius 50 pixels centered at ($X = 200$, $Y = 200$). A more portable program, of course, would have to determine the graphics hardware available and adjust these values accordingly.

*As with ANSI C, C++ allows functions to have a variable number of arguments, such as `float average(int number, ...)`, which can take one or more integer values. See Chapter 1, "The Turbo C++ language standard," in the Programmer's Guide for details.*

```
class Circle : public Point {   // Derived from class Point and
                                // ultimately from class Location
protected:
   int Radius;
public:
   Circle(int InitX = 200, int InitY = 200, int InitRadius = 50);
   void Show(void);
   void Hide(void);
   void Expand(int ExpandBy);
   void Contract(int ContractBy);
};
```

Now the declaration

```
Circle ACircle;
```

gives you a circle with the default center at (200,200) and radius 50. The declaration

```
Circle ACircle(50, 100);
```

gives a circle with center at 50, 100, with the default radius of 50.

The declaration

```
Circle ACircle(300)
```

gives a circle at $X = 300$, with default $Y = 200$ and radius = 50.

Any default arguments must be in consecutive rightmost positions in the argument list. For example, you couldn't declare

```
void func(int a = 10, int b, int c)
```

because the compiler wouldn't know which values are being supplied.

## More about overloading functions

*Overloading* is an important and pervasive concept in C++. When several different functions (whether member functions or ordinary) are defined with the same name within the same scope, they are said to be overloaded. You have met several such cases; for example, the three functions called **cube** on page 128. (Earlier versions of C++ required that such declarations be preceded by the keyword **overload**, but this is now obsolete.)

The basic idea is that overloaded function calls are distinguished by comparing the types of the actual arguments in the call and the formal argument signatures in the function definitions. The actual rules for disambiguation are beyond the scope of a primer and should rarely affect the beginner (who is hereby cautioned against the rash replication of function names). Among the possible complications are functions called with default actual arguments, or with a variable numbers of arguments; also, there are the normal C conversions of argument type to be considered, together with additional type conversions peculiar to C++. When faced with a call to a heavily overloaded function, the compiler tries to find a *best match*. If there is no best match, a compiler error results.

One of the most common cases is overloading a constructor so as to provide several different ways to create a new object of a class. To illustrate this, we will define a very simple *String* class. (For some fully functional string classes, refer to the books in the bibliography.)

*You can load and run STRING.CPP from the IDE. After running it, you'll have to activate the User Screen to see the output. Use the hot key **Alt-F5** or the **Window | User Screen** menu item.*

```
//STRING.CPP--Example from Chapter 5 of Getting Started */

#include <iostream.h>
#include <string.h>

class String {
   char *char_ptr;    // pointer to string contents
   int length;        // length of string in characters
public:
   // three different constructors
   String(char *text);           // constructor using existing string
   String(int size = 80);        // creates default empty string
   String(String& Other_String); // for assignment from another
                                 // object of this class
   ~String() {delete char_ptr;};
   int Get_len (void);
   void Show (void);
};

String::String (char *text)
{
   length = strlen(text);  // get length of text
   char_ptr = new char[length + 1];
   strcpy(char_ptr, text);
};

String::String (int size)
{
   length = size;
   char_ptr = new char[length+1];
   *char_ptr = '\0';
};

String::String (String& Other_String)
```

```
{
   length = Other_String.length;      // length of other string
   char_ptr = new char [length + 1];  // allocate the memory
   strcpy (char_ptr, Other_String.char_ptr); // copy the text
};

int String::Get_len(void)
{
   return (length);
};

void String::Show(void)
{
   cout << char_ptr << "\n";
};

main ()                               // test the functions
{
   String AString ("Allocated from a constant string.");
   AString.Show();

   String BString;            // uses default length
   cout << "\n" << BString.Get_len() << "\n" ; //display length
   BString = "This is BString";

   String CString(BString);   // invokes the third constructor
   CString.Show();            // note its contents
}
```

*When calling a constructor with no arguments (or when accepting all default arguments), don't put empty parentheses after the name of the object. For example, declare `String BString;`, not `String BString();`.*

The class **String** has three different constructors. The first takes an ordinary string constant such as "This is a string" and initializes a string with these contents. The second constructor uses a default length of 80, and allocates the string without storing any characters in it (this might be used to create a temporary buffer). Note that you can override the default simply by calling the constructor with a different length: Instead of declaring `String AString`, you could declare, for example, `String AString(40)`.

The third constructor takes a reference to another object of type **String** (recall that the ampersand after a type means a reference to that type, and is used to pass the address of a variable rather than a copy of its contents.) With this constructor you can now write statements such as these:

```
String AString("This is the first string"); // create and initialize
String BString = Astring;  // create then assign BString from AString
```

Note that constructors are involved in three related but separate aspects of an object's life story: creation, initialization, and assignment. The use of the = operator for class assignments leads us nicely to our next topic, operator overloading. Unless you define a special = operator for a class, C++ defaults to a member-by-member assignment.

## Overloading operators to provide new meanings

C++ has a special feature found in few other languages: existing operators such as **+** can be given new definitions to make them work in an appropriate, user-defined manner with your own class objects. Operators are a very concise way of doing business. If you didn't have them, an expression such as `line * width + pos` would have to be written something like this: `add(mult(line, width),pos)`. Fortunately, the arithmetic operators in C (and C++) already know how to work with all of the numeric data types—the same **+** that works with **int** values also works with **float**, for example. The same operator is used, but the code generated is clearly different, since integers and floating-point numbers are represented differently in memory. In other words, operators such as **+** are already overloaded, even in regular C. C++ simply extends this idea to allow user-defined versions of the existing operators.

*Whitespace is okay between the keyword* ***operator*** *and the operator symbol.*

To define an operator, you define a function that has as its name the keyword **operator** followed by the operator symbol. (So, for example, `operator+` names a new version of the **+** operator.) All operator functions are by definition overloaded: They use an operator that already has a meaning in C, but they redefine it for use with a new data type. The **+** operator, for example, already has the capability to add two values of any of the standard numeric types (**int**, **float**, **double**, and so on.)

Now we can add a **+** operator to the **String** class. This operator will concatenate two string objects (as in BASIC) returning the result as a string object with the appropriate length and contents. Since concatenating is "adding together," the **+** symbol is the appropriate one to use. The BASIC lobby often criticizes C for not having such natural string operations. With C++, you can go far beyond the built-in BASIC string facilities.

The file XSTRING.CPP, available on your distribution disks, has the following additions to STRING.CPP to provide a simple operator **+**.

```
//XSTRING.CPP--Example from Chapter 5 of Getting Started */
// version of STRING.CPP with overloaded operator +

#include <iostream.h>
#include <string.h>
```

```
class String {
   char *char_ptr;   // pointer to string contents
   int length;       // length of string in characters
public:
   // three different constructors
   String(char *text);          // constructor using existing string
   String(int size = 80);       // creates default empty string
   String(String& Other_String); // for assignment from another
                                // object of this class
   ~String() {delete char_ptr;}; // inline destructor
   int Get_len (void);
   String& operator+ (String& Arg);
   void Show (void);
};

String::String (char *text)
{
   length = strlen(text);  // get length of text
   char_ptr = new char[length + 1];
   strcpy(char_ptr, text);
};

String::String (int size)
{
   length = size;
   char_ptr = new char[length+1];
   *char_ptr = '\0';
};

String::String (String& Other_String)
{
   length = Other_String.length;       // length of other string
   char_ptr = new char [length + 1];   // allocate the memory
   strcpy (char_ptr, Other_String.char_ptr); // copy the text
};

String& String::operator+ (String& Arg)
{
  length += Arg.length;
  char * NewText = new char[length+1];
  strcpy(NewText, char_ptr);
  strcat(NewText, Arg.char_ptr);
  delete char_ptr;
  char_ptr = NewText;
  return *this;
}

int String::Get_len(void)
{
   return (length);
};
```

```
void String::Show(void)
{
   cout << char_ptr << "\n";
};

main ()                                     // test the functions
{
   String AString ("The Quick Brown fox");
   AString.Show();

   String BString(" jumps over Bill");
   String CString;
   CString = AString + BString;
   CString.Show();
}
```

When you run the program, *CString* is assigned the concatenation of the two strings *AString* and *BString*. So **CString.Show()** displays

*To see this display from the IDE, press Alt-F5 or Window \| User.*

```
The Quick Brown Fox jumps over Bill
```

The overloaded **+** takes only one explicit argument, so you may wonder how it manages to concatenate two strings. Well, the compiler treats the expression *AString* + *BString* as

```
AString.(operator +(BString))
```

so the **+** operator does access two string objects. The first is the **String** object currently being referenced, and the other is a second string object. The operator function adds the lengths of the two strings together, then uses the **strcat** library function to combine the contents of the two strings, which is then returned. This remarkable trick makes use of a "hidden" pointer known as **this**. What is **this**?

*__this__ is discussed in greater detail in the Programmer's Guide.*

Every call by a member function sets up a pointer to the object upon which the call is acting. This pointer can be referred via the keyword **this** (also known as "self" or rather "pointer-to-self" in OOP parlance), allowing functions to access the actual object. Now **this** is of type "pointer to String", so the return value must be ***this**, the actual current object, is exactly what is needed. Note, too, that individual members of the object involved in a function call can be referenced via the expression **this->member**. A further point to watch: **this** is available only to member functions, not to friend functions.

There are some restrictions when overloading operators:

- C++ can't distinguish between the prefix and postfix versions of **++** and **– –**.

- The operator you wish to define must already exist in the language. For example, you can't define the operator #.
- You can't overload the following operators:

  **. .* :: ?:**
- Overloaded operators keep their original precedence.
- If @ stands for any unary operator, the expressions **@x** and **x@** may be interpreted as either **x.operator@()** or as **operator@(x)**. If both forms have been declared, the compiler will try to resolve the ambiguity by matching the arguments. Similarly, with an overloaded binary operator, @, **x@y** could mean either **x.operator@(y)** or **operator@(x,y)**, and the compiler needs to look at the arguments if both forms have been defined. You saw an example of a binary operator in the string version of **+**, where `AString + BString` was interpreted as `AString.(operator +(BString))`.

## Friend functions

Normally, access to the private members of a class is restricted to member functions of that class. Occasionally it may be necessary to give outside functions access to the class's private data. The **friend** declaration within a class declaration lets you specify outside functions (or even outside classes) that will be granted access to the declared class's private members. You'll sometimes see an overloaded operator declared as a friend, but generally speaking friend functions are to be used sparingly—if their need persists in your project, it is often a sign that your class hierarchy needs revamping.

But, suppose that there is a fancy formatted printing function called *Fancy_Print* that you want to have access to the contents of your objects of class **String**. You can add the following line to the list of member function declarations:

*The position of the declaration doesn't matter.*

```
class String {
...
friend void Fancy_Print(String& AString);
...
```

In this admittedly artificial example, the *Fancy_Print* function can access the members *char_ptr* and *length* of objects of the **String** class. That is, if *AString* is a string object, *Fancy_Print* can access `AString.char_Ptr` and `AString.length`.

If the *Fancy_Print* function is a member of another class (for example, the class **Format**), use the scope resolution operator in the friend declaration:

```
friend void Format::Fancy_Print(String& AString);
```

You can also make a whole class the friend of the declared class, by using the word **class** in the declaration:

```
friend class Format;
```

Now any member function of the **Format** class can access the private members of the **String** class. Note that in C++, as in life, friendship is not transitive: if **X** is a friend of **Y**, and **Y** is a friend of **Z**, it does not follow that **X** is a friend of **Z**.

The friend declaration should be used only when it is really necessary; when without it you would have to have a convoluted class hierarchy. By its nature, the friend declaration diminishes encapsulation and modularity. In particular, if you find yourself wanting to make a whole class the friend of another class, consider instead the possibility of deriving a common derived class and using it to access the needed members.

## The C++ streams libraries

*This section is intended merely to whet your appetite and point you in the right direction. We encourage you to study the examples in Chapter 3, "C++ streams," in the Programmer's Guide and experiment on your own.*

While all the *stdio* library I/O functions (such as **printf** and **scanf**) are still available, C++ also provides a group of classes and functions for I/O defined in the **iostreams** library. To access these your program must have the directive `#include <iostream.h>`, as you may have noticed in some of our examples.

There are many advantages in using **iostreams** rather than *stdio*. The syntax is simpler, more elegant, and more intuitive. The C++ stream mechanism is also more efficient and flexible. Formatting output, for example, is simplified by extensive use of overloading. The same operator can be used to output both predefined and user-defined data types, avoiding the complexities of the **printf** argument list.

Starting with the stream as an abstraction for modeling any flow of data from a source (or producer) to a sink (or consumer), **iostream** provides a rich hierarchy of classes for handling buffered and unbuffered I/O for files and devices.

➡ Turbo C++ also supports the older (version 1.x) C++ **stream** library to assist programmers during the transition to the new **iostream** library of C++ release 2.0. If you have any C++ code that uses the obsolete **stream** classes, you can still maintain and run it with Turbo C++. However, given a choice, you should convert to the more efficient **iostream** and avoid **stream** when writing new code. Chapter 3, "C++ streams," in the *Programmer's Guide* explains the differences between the **stream** and **iostream** libraries, and provides some hints on conversion. See also OLDSTR.DOC on your distribution disks.

In this section we cover only the simpler classes in **iostream**. For a more detailed account, you should read Chapter 3, "C++ streams," in the *Programmer's Guide*. You can also browse through iostream.h on your distribution disks to see the many classes defined there and how they are derived using both single and multiple inheritance.

## Standard I/O

C++ provides four predefined stream objects defined as follows:

| | |
|---|---|
| ■ **cin** | standard input, usually the keyboard, corresponding to **stdin** in C |
| ■ **cout** | standard output, usually the screen, corresponding to **stdout** in C |
| ■ **cerr** | standard error output, usually the screen, corresponding to **stderr** in C |
| ■ **clog** | a fully-buffered version of **cerr** (no C equivalent) |

You can redirect these standard streams from and to other devices and files. (In C, you can redirect only **stdin** and **stdout**.) You have already seen the most common of these, **cin** and **cout**, in some of the examples in this chapter.

A simplified view of the **iostream** hierarchy, from primitive to specialized, is as follows:

| | |
|---|---|
| ■ **streambuf** | provides methods for memory buffers |
| ■ **ios** | handles stream state variables and errors |
| ■ **istream** | handles formatted and unformatted character conversions *from* a **streambuf** |
| ■ **ostream** | handles formatted and unformatted character conversions *to* a **streambuf** |
| ■ **iostream** | combines **istream** and **ostream** to handle bidirectional operations on a single stream |

| | |
|---|---|
| ■ **istream_withassign** | provides constructors and assignment operators for the **cin** stream. |
| ■ **ostream_withassign** | provides constructors and assignment operators for the **cout, cerr** and **clog** streams. |

*<< used with streams is called the insertion or put to operator, while >> is called the extraction or get from operator.*

The **istream** class includes overloaded definitions for the **>>** operator for the standard types [**int, long, double, float, char,** and **char*** (string)]. Thus the statement `cin >> x;` calls the appropriate **>>** operator function for the **istream cin** defined in iostream.h and uses it to direct this input stream into the memory location represented by the variable *x*. Similarly, the **ostream** class has overloaded definitions for the **<<** operator, which allows the statement `cout << x;` to send the value of *x* to `ostream cout` for output.

These operator functions return a reference to the appropriate stream class type (e.g., **ostream&**) in addition to moving the data. This allows you to chain several of these operators together to output or input sequences of characters:

```
int i=0, x=243; double d=0;
cout << "The value of x is " << x << '\n';
cin >> i >> d; // key an int, space, then a double
```

The second line would display "The value of x is 243" followed by a new line. The next statement would ignore whitespace, read and convert the keyed characters to an integer and place it in *i*, ignore following whitespace, read and convert the next keyed characters to a **double** and place it in *d*.

The following program simply copies **cin** to **cout**. In the absence of redirection, it copies your keyboard input to the screen:

*This program simply stores each input character in the variable ch and then outputs the value of ch to the screen.*

```
// COPYKBD.CPP      Copies keyboard input to screen

#include <iostream.h>

main()
{
   char ch;
   while (cin >> ch)
      cout << ch;
}
```

Note how you can test *(cin >> ch)* as a normal Boolean expression. This useful trick is made possible by definitions in the class **ios**. Briefly, an expression such as *(cout)* or *(cin >> ch)* is cast as a pointer, the value of which depends on the error state of the stream. A null pointer (tested as false) indicates an error in the stream, while a non-null pointer (tested as true) means no errors. You can also reverse the test using **!**, so that *(!cout)* is true for an error in the **cout** stream and false if all is well:

```
if (!cout) errmsg("Output error!");
```

## Formatted output

Simple I/O in C++ is efficient because only minimal conversion is done according to the data type involved. For integers, conversion is the same as the default for **printf**. The statements

```
int i=5; cout << i;
```

and

```
int i=5; printf("%d",i);
```

give the same result.

Formatting is determined by a set of format state flags enumerated in **ios**. These determine, for each active stream, the conversion base (decimal, octal, and hexadecimal), padding left or right, the floating-point format (scientific or fixed), and whether whitespace is to be skipped on input. Other parameters you can vary include field width (for output) and the character used for padding. These flags can be tested, set, and cleared by various member functions. The following snippet shows how the functions **ios::width** and **ios::fill** work:

```
int previous_width, i = 87;
previous_width = cout.width(7); // set field width to 7
                                // and save previous width
cout.fill('*');                 // set fill character to *
cout << i << '\n';              // display *****87 <newline>
// after << the width is cleared to 0
// previous width may have been set without a subsequent <<
// so you may want to restore it with the following line.
cout.width(previous_width);
```

Setting *width* to zero (the default) means that the display will take as many screen positions as needed. If the given width is insufficient for the correct representation, a width of zero is assumed (that is, there is no truncation). Default padding gives right justification (left padding) for all types.

**setf** and **unsetf** are two general functions for setting and clearing format flags:

```
cout.setf(ios::left, ios::adjustfield);
```

This sets left padding. The first argument uses enumerated mnemonics for the various bit positions (possibly combined using **&** and **|**) and the second argument is the target field in the format state. **unsetf** works the same way but clears the selected bits. (More on these in Chapter 3, "C++ streams," in the *Programmer's Guide.*)

### Manipulators

A rather more elegant way of setting the format flags (and performing other stream chores) uses special mechanisms known as *manipulators*. Like the **<<** and **>>** operators, manipulators can be embedded in a chain of stream operations:

```
cout << setw(7) << dec << i << setw(6) << oct << j;
```

Without manipulators, this would take six separate statements.

The *parameterized manipulator* **setw** takes a single **int** argument to set the field width.

The non-parameterized manipulators, such as **dec, oct**, and **hex,** set the conversion base to decimal, octal, and hexadecimal. In the above example, `int i` would display in decimal on a field of width 7; `int j` would display in octal on a field of width 6.

Other simple parameterized manipulators include **setbase, setfill, setprecision, setiosflags**, and **resetiosflags**. To use any of the parameterized manipulators, your program must have include both of these header files: iomanip.h and iostream.h. Non-parameterized manipulators do not require **iomanip.h**.

Useful non-parameterized manipulators include:

**ws** (whitespace extractor): `istream >> ws;` will discard any whitespace in **istream**.

**endl** (endline and flush): `ostream << endl;` will insert a newline in **ostream**, then flush the **ostream**.

**ends** (end string with null): `ostream << ends;` will append a null to **ostream**.

**flush** (flush output stream): `ostream << flush;` flushes the **ostream**.

### put, write, and get

Two general output functions are worthy of mention: **put** and **write**, declared in **ostream** as follows:

```
ostream& ostream::put(char ch);
// send ch to ostream

ostream& ostream::write(const char* buff, int n);
// send n characters from buff to ostream; watch the size of n!
```

**put** and **write** let you output unformatted binary data to an **ostream** object. **put** outputs a single character, while **write** can send any number of characters from the indicated buffer. **write** is useful when you want to output raw data that may include nulls. (Note that writing binary data requires that the file be opened in binary mode.) The normal string extractor would not work since it terminates on a null.

The input version of **put** is called **get**:

```
char ch;
cin.get(ch);
// grab next char from cin whether whitespace or not
```

Another version of **get** lets you grab any number of raw, binary characters from an **istream**, up to a designated maximum, and place them in a designated buffer (as with **write**, files must be opened in binary mode):

```
istream& istream::get(char *buf, int max, int term='\n');
// read up to max chars from istream, and place them in buf. Stop if
// term char is read.
```

You can set *term* to a specific terminating character (the default is the newline character), at which **get** will stop if reached before *max* characters have been transferred to *buf*.

## Disk I/O

The **iostream** library includes many classes derived from **streambuf**, **ostream**, and **istream**, thereby allowing a wide choice of file I/O methods. The **filebuf** class, for example, supports I/O through file descriptors with member functions for opening, closing, and seeking. Contrast this with the class **stdiobuf** that supports I/O via **stdio** FILE structures, allowing some compatibility when you need to mix C and C++ code.

The most generally useful classes for the beginner are **ifstream** (derived from **istream**), **ofstream** (derived from **ostream**), and **fstream** (derived from **iostream**). These all support formatted file I/O using **filebuf** objects. Here's a simple example that copies an existing disk file to another specified file:

*This code is available as DCOPY.CPP.*

```
/* DCOPY.CPP -- Example from Chapter 5 of Getting Started */
/* DCOPY source-file destination-file                  *
 * copies existing source-file to destination-file     *
 * If latter exists, it is overwritten; if it does not *
 * exist, DCOPY will create it if possible             *
 */

#include <iostream.h>
#include <process.h>    // for exit()
#include <fstream.h>    // for ifstream, ofstream

main(int argc, char* argv[])  // access command-line arguments
{
   char ch;
   if (argc != 3)      // test number of arguments
   {
      cerr << "USAGE: dcopy file1 file2\n";
      exit(-1);
   }

   ifstream source;    // declare input and output streams
   ofstream dest;

   source.open(argv[1],ios::nocreate); // source file must be there
   if (!source)
   {
      cerr << "Cannot open source file " << argv[1] <<
          " for input\n";
      exit(-1);
   }
   dest.open(argv[2]);   // dest file will be created if not found
          // or cleared/overwritten if found
   if (!dest)
   {
      cerr << "Cannot open destination file " << argv[2] <<
         " for output\n";
      exit(-1);
   }

   while (dest && source.get(ch)) dest.put(ch);

   cout << "DCOPY completed\n";

   source.close();        // close both streams
   dest.close();
}
```

Note first that `#include <fstream>` also pulls in iostream.h. DCOPY uses the standard method of accessing command-line arguments to check whether the user specified the two files involved. When this argument list is used with the **main** function, the argument *argc* contains the number of command-line arguments (including the name of the program itself), and the strings *argv[1]* and *argv[2]* contain the two file names entered. A typical command-line invocation of this program would be

```
dcopy letter.spr letter.bak
```

To see how DCOPY works, examine the following lines:

```
ifstream source;   // declare an input stream (ifstream object)
  ...
open.source(argv[1],ios::nocreate); // source file must be there
```

The declaration invokes a constructor of **ifstream** (the class for handling input file streams) to create a stream object called *source*. Before we can make use of *source*, we must create a file buffer and associate the stream and buffer with a real, physical file. Both tasks are performed by the member function **open** in **ifstream**. The **open** function needs a file name string and, optionally, one or two other arguments to specify the mode and protection rights. The file name here is given as *argv[1]*, namely, the source file supplied in the command line.

A neater alternative to the above declaration is:

```
ifstream source(argv[1],ios::nocreate); // source file must be there
// this creates source and opens the file as well
```

The mode argument **ios::nocreate** tells **open** not to create a file if the named file is not found. For DCOPY, we clearly want **open** to fail if the named source file is not on the disk. Later, you'll see the other mode arguments available. If the file *argv[1]* cannot be opened for any reason (usually because the file is not found), the value of *source* is effectively set to zero (false), so that `(!source)` tests true, giving us an error message, then exiting.

In fact, we could determine the possible reason for the failure to open the source file by examining the error bits set in the *stream state*. The member functions **eof**, **fail**, and **bad** test various error bits and return true if they are set. Alternatively, **rdstate** returns the error state in an **int**, and you can then test which bits are set. The **eof** (end of file) is not really an error *per se*, but it needs to be tested and acted upon since a stream cannot be usefully accessed beyond its final character. Note that once a stream is in an error state (including **eof**), no further I/O is permitted. The function **clear** is provided for clearing some or all error bits, allowing you to resume after clearing a nonfatal situation.

Back in DCOPY.CPP, if all is well with the source file, we then try to open the destination file with the **ofstream** object, *dest*. With output files, the default situation is that a file will be created if it does not exist; if it exists it will be cleared and recreated as an empty file. You can modify this behavior by adding a second argument, *mode*, to the declaration of *dest*. For example:

```
ofstream dest(argv[2],ios::app|ios::nocreate);
```

will try to open *dest* in *append* mode, failing if *dest* is *not* found. In append mode, the data in the source file would be added to the end of *dest*, leaving the previous contents undisturbed. Other mode flags enumerated in class **ios** (note the scope operator in **ios::app**), are **ate** (seek to end of file); **in** (open for input, used with **fstreams**, since they can be opened for both input and output); **out** (open for output, also used with **fstreams**); **trunc** (discard contents if file exists); **noreplace** (fail if file exists).

Once both files have been opened, the actual copying is achieved in typically condensed C fashion. Consider the Boolean expression tested by the **while** loop:

*When C tests `(x && y)`, it will not bother to test y if x proves false. Since dest is less likely to "fail" than source.get(ch), you might consider reversing the entries.*

```
(dest && source.get(ch))
```

We have seen that *dest* will test true until an error occurs. Similarly the call *source.get(ch)* will test true until either a reading error occurs or until the end of the file is reached. In the absence of "hard" errors, then, the loop **get**s characters from *source* and **put**s them in *dest* until an end of file situation makes *source* false.

There are many more file I/O features in the **iostream** library. And **iostream** can also help you with in-memory formatting, where your streams are in RAM. Special classes, such as **strstreambuf**, are provided for in-memory stream manipulation.

## I/O for user-defined data types

A real benefit with C++ streams is the ease with which you can overload **>>** and **<<** to handle I/O for your own personal data types. Consider a simple data structure that you may have declared:

```
struct emp {
   char *name;
   int dept;
   long sales;
};
```

To overload **<<** to output objects of type *emp*, you need the following definition:

```
ostream& operator << (ostream& str, emp& e)
{
str << setw(25) << e.name << ": Department " << setw(6) << e.dept <<
<< tab << " Sales $" << e.sales << '\n';
return str;
}
```

Note that the operator-function **<<** must return **ostream&**, a reference to **ostream**, so that you can chain your new **<<** just like the predefined insertion operator. You can now output objects of type *emp* as follows:

```
#include <iostream.h>
#include <iomanip.h>            // don't forget this!
...
emp jones = {"S. Jones", 25, 1000};
cout << jones;
```

giving the display

```
S. Jones: Department 25     Sales $1000
```

Did you spot the manipulator **tab** in the **<<** definition? This is not a standard manipulator—but a user-defined one:

```
ostream& tab(ostream& str) {
   return str << '\t';
}
```

This, of course, is trivial, but nevertheless makes for more legible code.

An input routine for *emp* can be similarly devised by overloading **>>**. This is left as an exercise for the reader.

## Where to now?

A suggestion for your first C++ project is to take the FIGURES module shown on page 163 (you have it on disk) and extend it. Points, circles, and arcs are by no means enough. Create objects for lines, rectangles, and squares. When you're feeling more ambitious, create a pie-chart object using a linked list of individual pie-slice figures.

One more subtle challenge is to implement classes to handle relative position. A relative position is an offset from some base point, expressed as a positive or negative difference. A point at relative coordinates –17,42 is 17 pixels to the left of the base point, and 42 pixels down from that base point. Relative positions are necessary to combine figures effectively into single larger figures, since multiple-figure combinations cannot always be tied together at each figure's anchor point. Better to define an *RX* and *RY* field in addition to anchor point *X,Y*, and have the final position of the object onscreen be the sum of its anchor point and relative coordinates.

Once you feel comfortable with C++, start building its concepts into your everyday programming chores. Take some of your more useful existing utilities and rethink them in C++ terms. Try to see the classes in your hodgepodge of function libraries—then rewrite the functions in class form. You'll find that libraries of classes 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 a class 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

C++ 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. C++ 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 you not only have a toolkit—you have tools to build new tools!

C H A P T E R

6

# Hands-on C++

*This chapter is a concise, hands-on tutorial for C++.*

In order to give you a sense of how C++ looks and how to accomplish tasks in C++, this chapter moves quickly through a large number of concepts with a minimum of verbiage. It is intended to be used as you work at your computer; you can load and run each of these programs (which are in your EXAMPLES subdirectory, along with any header and other files that you'll need). If you want a more in-depth treatment of C++, especially of the concepts underlying object-oriented programming, read Chapter 5, "A C++ primer." You might also want to refer to Chapter 1, "The Turbo C++ language standard," in the *Programmer's Guide* for precise details about the syntax and use of C++.

***Important!***

In this chapter, we assume that you are familiar with the C language, and that you know how to compile, link, and execute a source program with Turbo C++. We start with simple examples that grow in complexity so that new concepts will stand out. It is reasonable that such examples will not be bulletproof (in other words, they don't check for memory failure and so on). This chapter is not a treatise on data structures or professional programming techniques; instead, it is a gentle introduction to a complicated language.

This chapter is divided into two sections. The first section provides C++ alternatives to C programming knowledge and habits you might have. The second section provides a swift introduction to the kernel of C++: Object-oriented programming using classes and inheritance.

## A better C: Making the transition from C

*When referring to line numbers, we've counted blank lines.*

Although knowing C is helpful to learning C++, sometimes that knowledge can get in the way, particularly in the areas that aren't specifically object-oriented programming, yet where C++ does things differently from C. For that reason, this section shows how to accomplish in C++ many of the same kinds of actions you would perform in C: writing text to the screen, commenting your code, creating and using constants, working with stream I/O and inline functions, and so on.

### Program 1

*Source*

```
// ex1.cpp:   A First Glance
// from Chapter 6 of Getting Started
#include <iostream.h>

main()
{
   cout << "Frankly, my dear...\n";
   cout << "C++ is a better C.\n";
}
```

*Output*

```
Frankly, my dear...
C++ is a better C.
```

Note the new comment syntax in the first line of this program. All characters from the first occurrence of double slashes to the end of a line are considered a comment, although you can still use the traditional **/*...*/** style. File names which have a .CPP extension are assumed to be C++ files (or you could use the command-line compiler option **–P**).

The third line includes the standard header file **iostream.h**, which replaces much of the functionality of **stdio.h**. **cout** is an *output stream*, and is used to send characters to standard output (as **stdout** does in C). The **<<** operator (pronounced "put to") sends the data on its right to the stream on its left. The context of the **<<** operator here distinguishes it from the arithmetic shift-left operator, which uses the same symbol. (Such multiple use of operators and functions is quite common in C++ and is called *overloading*.)

## Program 2

*Source*

```
// ex2.cpp:   An interactive example
// from Chapter 6 of Getting Started
#include <iostream.h>

main()
{
   char name[16];
   int age;

   cout << "Enter your name: ";
   cin >> name;
   cout << "Enter your age: ";
   cin >> age;

   if (age < 21)
      cout << "You young whippersnapper, " << name << "!\n";
   else if (age < 40)
      cout << name << ", you're still in your prime!\n";
   else if (age < 60)
      cout << "You're over the hill, " << name << "!\n";
   else if (age < 80)
      cout << "I bow to your wisdom, " << name << "!\n";
   else
      cout << "Are you really " << age << ", " << name << "?\n";
}
```

*Sample execution*

```
Enter your name: Don
Enter your age: 40
You're over the hill, Don!
```

**cin** is an input stream connected to standard input. It can correctly process all the standard data types. You may have noticed in C that printing a prompt without a newline character to **stdout** required a call to **fflush(stdout)** in order for the prompt to appear. In C++, whenever **cin** is used it flushes, **cout** automatically (you can turn this automatic flushing off if it's on by default).

## Program 3

*Source*

*In an important change from C, declarations can appear anywhere a statement can.*

```
// ex3.cpp:   Inline Functions
// from Chapter 6 of Getting Started
#include <iostream.h>

const float Pi = 3.1415926;

inline float area(const float r) {return Pi * r * r;}

main()
```

```
{
   float radius;

   cout << "Enter the radius of a circle: ";
   cin >> radius;
   cout << "The area is " << area(radius) << "\n";
}
```

*Sample execution*

```
Enter the radius of a circle: 3
The area is 28.274334
```

A constant identifier behaves like a normal variable (that is, its scope is the block that defined it, and it is subject to type checking) except that it cannot appear on the left-hand side of an assignment statement (or anywhere an lvalue is required). Using **#define** is *almost* obsolete in C++.

The keyword **inline** tells the compiler to insert code directly whenever possible, in order to avoid the overhead of a function call. In all other ways (scope, etc.) an inline function behaves like a normal function. Its use is recommended over **#define**d macros (except, of course, where you depend on the macro-substitution tricks of the preprocessor). This feature is intended for simple, one-line functions.

## Program 4

*Source*

```
// ex4.cpp:   Default arguments and Pass-by-reference
// from Chapter 6 of Getting Started
#include <iostream.h>
#include <ctype.h>

int get_word(char *, int &, int start = 0);

main()
{
   int word_len;
   char *s = "  These words will be printed one-per-line  ";

   int word_idx = get_word(s,word_len);          // line 13
   while (word_len > 0)
   {
      cout.write(s+word_idx, word_len);
              cout << "\n";
      //cout << form("%.*s\n",word_len,s+word_idx);
      word_idx = get_word(s,word_len,word_idx+word_len);
   }
}
```

*It's good programming style to make null loop bodies stand out.*

```
int get_word(char *s, int& size, int start)
```

```
{
   // Skip initial whitespace
   for (int i = start; isspace(s[i]); ++i);
   int start_of_word = i;

   // Traverse word
   while (s[i] != '\0' && !isspace(s[i]))
      ++i;
   size = i - start_of_word;
   return start_of_word;
}
```

*Output*

```
These
words
will
be
printed
one-per-line
```

The prototype for the function **get_word** in the sixth line has two special features. The second argument is declared to be a *reference* parameter. This means that the value of that argument will be modified in the calling program (this is equivalent to **var** parameters in Pascal, and is accomplished through pointers in C). By this means, the variable **word_len** is updated in **main**, and yet we can still return another useful value with the function **get_word**.

*One exciting feature of C++ is the **default argument**.*

The third argument is a *default* argument. This means that it can be omitted (as in line 13), in which case the value of 0 is passed automatically. Note that the default value need only be specified in the first mention of the function. Only the trailing arguments of a function can supply default values.

## Object support

The world is made up of things that both possess *attributes* and exhibit *behavior*. C++ provides a model for this by extending the notion of a structure to contain functions as well as data members. This way an object's complete identity is expressed through a single language construct. The notion of object-oriented support then is more than a notational convenience—it is a tool of thought.

## Program 5

Suppose we want to have an online dictionary. A dictionary is made up of definitions for words. We will first model the notion of a definition.

*You'll need to compile DEF.CPP to an OBJ file, then link it in with either EX6.CPP or EX7.CPP (or load EX5.PRJ). You might also want to compile it with Debug Info checked so you can step through and watch the program flow.*

```
// def.h:   A word definition class
// from Chapter 6 of Getting Started
#include <string.h>

const int Maxmeans = 5;

class Definition
{
   char *word;                     // Word being defined
   char *meanings[Maxmeans];       // Various meanings of this word
   int nmeanings;

public:
   void put_word(char *);
   char *get_word(char *s) {return strcpy(s,word);};  // line 15
   void add_meaning(char *);
   char *get_meaning(int, char *);
};
```

*In other object-oriented languages, classes are often called **objects**, and member functions are called **methods**.*

In traditional C style, we put definitions in an include file. The keyword **class** introduces the object description. By default, members of a class are private (though you can explicitly use the keyword **private**), so in this case the fields in lines 9 through 11 can only be accessed by functions of the class. (In C++, class functions are called *member functions*.) To make these functions available as a user interface, they are preceded by the keyword **public**. Note that the **inline** keyword is not required inside class definitions (line 15).

The implementation is usually kept in a separate file:

```
// def.cpp:   Implementation of the Definition class
// from Chapter 6 of Getting Started
#include <string.h>
#include "def.h"

void Definition::put_word(char *s)
{
   word = new char[strlen(s)+1];
   strcpy(word,s);
   nmeanings = 0;
}

void Definition::add_meaning(char *s)
```

```
{
   if (nmeanings < Maxmeans)
   {
      meanings[nmeanings] = new char[strlen(s)+1];
      strcpy(meanings[nmeanings++],s);
   }
}

char * Definition::get_meaning(int level, char *s)
{
   if (0 <= level && level < nmeanings)
      return strcpy(s,meanings[level]);
   else
      return 0;                                  // line 27
}
```

The *scope resolution operator* (::) informs the compiler that we are defining member functions for the **Definition** class (it's good practice to capitalize the first letter of a class to avoid name conflicts with library functions). The keyword **new** in line 8 is a replacement for the dynamic memory allocation function **malloc**. In C++, by convention, zero is used instead of NULL for pointers (line 27). Although we didn't do so here, it is advisable to verify that **new** returns a non-zero value.

*Source*

```
// ex5.cpp:  Using the Definition class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "def.h"

main()
{
   Definition d;          // Declare a Definition object
   char s[81];

   // Assign the meanings
   d.put_word("class");
   d.add_meaning("a body of students meeting together to \
study the same subject");
   d.add_meaning("a group sharing the same economic status");
   d.add_meaning("a group, set or kind sharing the same attributes");

   // Print them
   cout << d.get_word(s) << ":\n\n";
   for (int i = 0; d.get_meaning(i,s) != 0; ++i)
      cout << i+1 << ": " << s << "\n";
}
```

*Output*

```
class:

1: a body of students meeting together to study the same subject
```

```
2: a group sharing the same economic status
3: a group, set, or kind sharing the same attributes
```

## Program 6

*From the command line, build DICTION.OBJ and DEF.OBJ with EX6.CPP. From the IDE, use the EX6.PRJ project file.*

We can now define a dictionary as a collection of definitions.

```
// diction.h:   The Dictionary class
// from Chapter 6 of Getting Started
#include "def.h"

const int Maxwords = 100;

class Dictionary
{
   Definition *words;       // An array of definitions; line 9
   int nwords;

   int find_word(char *);   // line 12

public:
   // The constructor is on the next line
   Dictionary(int n = Maxwords)
      {nwords = 0; words = new Definition[n];};
   ~Dictionary() {delete words;};
   void add_def(char *s, char **def); // The destructor; line 17
   int get_def(char *, char **);
};
```

The function **find_word** on line 12 is for internal use only by the **Dictionary** class and so is kept private. A function with the same name as the class is called a *constructor* (line 16). It is called once whenever an object is declared. It is used to perform initializations; here we are dynamically allocating space for an array of definitions. A *destructor* (line 17) is called whenever an object goes out of scope (in this case, the **delete** operator will free the memory previously allocated by the constructor). In order to have an array of member objects (line 9), the included class must either have a constructor with no arguments or no constructor at all (the **Definition** class has none).

```
// diction.cpp:   Implementation of the Dictionary class
// from Chapter 6 of Getting Started
#include "diction.h"

int Dictionary::find_word(char *s)
{
   char word[81];
   for (int i = 0; i < nwords; ++i)
```

```
         if (stricmp(words[i].get_word(word),s) == 0)
            return i;

      return -1;
   }

   void Dictionary::add_def(char *word, char **def)
   {
      if (nwords < Maxwords)
      {
         words[nwords].put_word(word);
         while (*def != 0)
            words[nwords].add_meaning(*def++);
         ++nwords;
      }
   }

   int Dictionary::get_def(char *word, char **def)
   {
      char meaning[81];
      int nw = 0;
      int word_idx = find_word(word);
      if (word_idx >= 0)
      {
         while (words[word_idx].get_meaning(nw,meaning) != 0)
         {
            def[nw] = new char[strlen(meaning)+1];
            strcpy(def[nw++],meaning);
         }
         def[nw] = 0;
      }

      return nw;
   }
```

We can now use the **Dictionary** class without any reference to the **Definition** class (the output is the same as in the previous example).

*Source*

```
// ex6.cpp:   Using the Dictionary class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "diction.h"

main()
{
   Dictionary d(5);
   char *word = "class";
   char *indef[4] =
     {"a body of students meeting together to study the same",
      "subject a group sharing the same economic status",
         "a group, set or kind sharing the same attributes",
         0);
      char *outdef[4];

      d.add_def(word,indef);
      cout << word << ":\n\n";
      int ndef = d.get_def(word,outdef);
      for (int i = 0; i < ndef; ++i)
         cout << i+1 << ": " << outdef[i] << "\n";
}
```

In the **Dictionary** implementation, we specifically called the **Definition** member functions. Sometimes it is desirable to allow certain functions or even an entire class to have access to the private members of another. We could declare the **Dictionary** class to be a **friend** to the **Definition** class (line 18):

*Build LIST.OBJ with EX7.CPP*

```
// def2.h:   A word definition class
// from Chapter 6 of Getting Started
#include <string.h>

const int Maxmeans = 5;

class Definition
{
   char *word;                        // Word being defined
   char *meanings[Maxmeans];          // Various meanings of this word
   int nmeanings;

public:
   void put_word(char *);
   char *get_word(char *s) {return strcpy(s,word);};
   void add_meaning(char *);
   char *get_meaning(int, char *);
   friend class Dictionary;           // line 18
};
```

The implementation of **find_word** could then access **Definition** members directly (line 5 in the following code):

```
int Dictionary::find_word(char *s)
{
   char word[81];
   for (int i = 0; i < nwords; ++i)
      if (stricmp(words[i].word),s) == 0)
         return i;

   return -1;
}
```

## Program 7

One of the key features of object-oriented programming is *inheritance*. A new class can inherit the data and member functions of an existing ("base") class (the new class is said to be *derived* from the base class). In this program, we define **List**, a base class for processing a list of integers, then derive **Stack**, a class to handle a stack (which is a special kind of list). First, we create the header file:

*To try these out, build LIST.OBJ and EX7. CPP, or use EX7.PRJ.*

```
// list.h:   A Integer List Class
// from Chapter 6 of Getting Started
const int Max_elem = 10;

class List
{
   int *list;          // An array of integers
   int nmax;           // The dimension of the array
   int nelem;          // The number of elements

public:
   List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
   ~List() {delete list;};
   int put_elem(int, int);
   int get_elem(int&, int);
   void setn(int n) {nelem = n;};
   int getn() {return nelem;};
   void incn() {if (nelem < nmax) ++nelem;};
   int getmax() {return nmax;};
   void print();
};
```

Then we create the source code:

```
// list.cpp:  Implementation of the List Class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "list.h"

int List::put_elem(int elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      list[pos] = elem;    // Put an element into the list
      return 0;
   }
   else
      return -1;           // Non-zero means error
}
```

```
int List::get_elem(int& elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      elem = list[pos];     // Retrieve a list element
      return 0;
   }
   else
      return -1;            // non-zero means error
}

void List::print()
{
   for (int i = 0; i < nelem; ++i)
      cout << list[i] << "\n";
}
```

And finally we use the new class:

```
// ex7.cpp:   Using the List class
// from Chapter 6 of Getting Started
#include "list.h"

main()
{
   List l(5);
   int i = 0;

   // Insert the numbers 1 through 5
   while (l.put_elem(i+1,i) == 0)
      ++i;
   l.setn(i);

   l.print();
}
```

*Output*

```
1
2
3
4
5
```

## Program 8

*Build STACK.OBJ and LIST.OBJ with EX8.CPP, or use EX8.PRJ.*

```
// stack2.h:   A Stack class derived from the List class
// from Chapter 6 of Getting Started
#include "list2.h"

class Stack : public List                // line 5
{
   int top;
public:
   Stack() {top = 0;};
   Stack(int n) : List(n) {top = 0;};      // line 11
   int push(int elem);
   int pop(int& elem);
   void print();
};
```

To define a derived class, the base class definition must be available, so we include its header file (line 3). Line 5 informs the compiler that the **Stack** class is derived from the **List** class. The keyword **public** states that the public members of **List** should be considered public in **Stack** also (this is what is usually needed). Since the **List** class has a constructor that takes an argument, the *Stack* constructor invokes the **List** constructor directly (line 11). Base class constructors are executed before those of a derived class.

```
// stack.cpp:   Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack.h"

int Stack::push(int elem)
{
   int m = getmax();
   if (top < m)
   {
      put_elem(elem,top++);
      return 0;
   }
   else
      return -1;
}

int Stack::pop(int& elem)
{
   if (top > 0)
   {
      get_elem(elem,--top);
      return 0;
   }
   else
      return -1;
}

void Stack::print()
{
   int elem;

   for (int i = top-1; i >= 0; --i)
```

```
      {  // Print in LIFO order
         get_elem(elem,i);
         cout << elem << "\n";
      }
   }
```

Note that the public member functions of the **List** class can be used directly, because a **Stack** is a **List**. However, the private members of the **List** portion of a **Stack** object cannot be referenced directly.

```
// ex8.cpp:   Using the Stack Class
// from Chapter 6 of Getting Started
#include "stack.h"

main()
{
   Stack s(5);
   int i = 0;

   // Insert the numbers 1 through 5
   while (s.push(i+1) == 0)
      ++i;

   s.print();
}
```

*Output*

```
5
4
3
2
1
```

## Program 9

Sometimes it is convenient to allow a derived class to have direct access to some of the private data members of a base class. Such data members are said to be *protected*.

*Build EX9.CPP, LIST2.OBJ, STACK2.OBJ, or use EX9.PRJ*

```
// list2.h:   A Integer List Class
// from Chapter 6 of Getting Started
const int Max_elem = 10;

class List
{
protected:      // The protected keyword gives subclasses
                // direct access to inherited members
   int *list;         // An array of integers
   int nmax;          // The dimension of the array
   int nelem;         // The number of elements
public:
   List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
   ~List() {delete list;};
   int put_elem(int, int);
   int get_elem(int&, int);
   void setn(int n) {nelem = n;};
   int getn() {return nelem;};
   void incn() {if (nelem < nmax) ++nelem;};
   int getmax() {return nmax;};
   virtual void print();                  // line 22
};
```

We can now replace calls to **List's** member functions with direct references to **List's** data in the **Stack** implementation.

```
// stack.cpp:   Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack.h"

int Stack::push(int elem)
{
   int m = getmax();
   if (top < m)
   {
      put_elem(elem,top++);
      return 0;
   }
   else
      return -1;
}

int Stack::pop(int& elem)
{
   if (top > 0)
   {
      get_elem(elem,--top);
      return 0;
   }
   else
      return -1;
}

void Stack::print()
{
   int elem;

   for (int i = top-1; i >= 0; --i)
   {  // Print in LIFO order
      get_elem(elem,i);
      cout << elem << "\n";
      }
   }
```

And then we can try it out:

```
// ex9.cpp:  Using the print() virtual function
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack2.h"

main()
{
   Stack s(5);
   List l, *lp;
   int i = 0;

   // Insert the numbers 1 through 5 into the stack
   while (s.push(i+1) == 0)
      ++i;

   // Put a couple of numbers into the list
   l.put_elem(1,0);
   l.put_elem(2,1);
   l.setn(2);

   cout << "Stack:\n";
   lp = &s;            // line 22
   lp->print();        // Invoke the Stack print() method; line 23

   cout << "\nList:\n";
   lp = &l;
   lp->print();        // Invoke the List print() method; line 27
}
```

*Output*

```
Stack:
5
4
3
2
1

List:
1
2
```

The above example illustrates *polymorphism* (also known as "late binding" or "dynamic binding," which in C++ is accomplished using *virtual functions*). This means that an object's type is not identified until run time. By defining the **print** member function to be **virtual** (see line 22 of "list2.h"), we can invoke the different **print** member functions through a pointer to the base class. In line 22 above, *lp* points to a **Stack** object (remember: a **Stack** is a **List**), so the **Stack** print method is invoked in line 23. Likewise, the **List print** member function is executed in line 27.

## Summary

There is much more to C++ than this chapter covers (multiple inheritance, for example). As stated at the beginning, this chapter is intended give you a sense of the "look and feel" of C++, to show how it differs from C, and to demonstrate how to use most of the basic features of C++. For more information on the basic concepts of C++, read or review Chapter 5, "A C++ primer;" Chapter 1, "The Turbo C++ language standard," in the *Programmer's Guide* gives more advanced material on C++. And check the bibliography; it provides a list of books on C++, including books specific to Turbo C++.

C H A P T E R

7

# Debugging in the new IDE

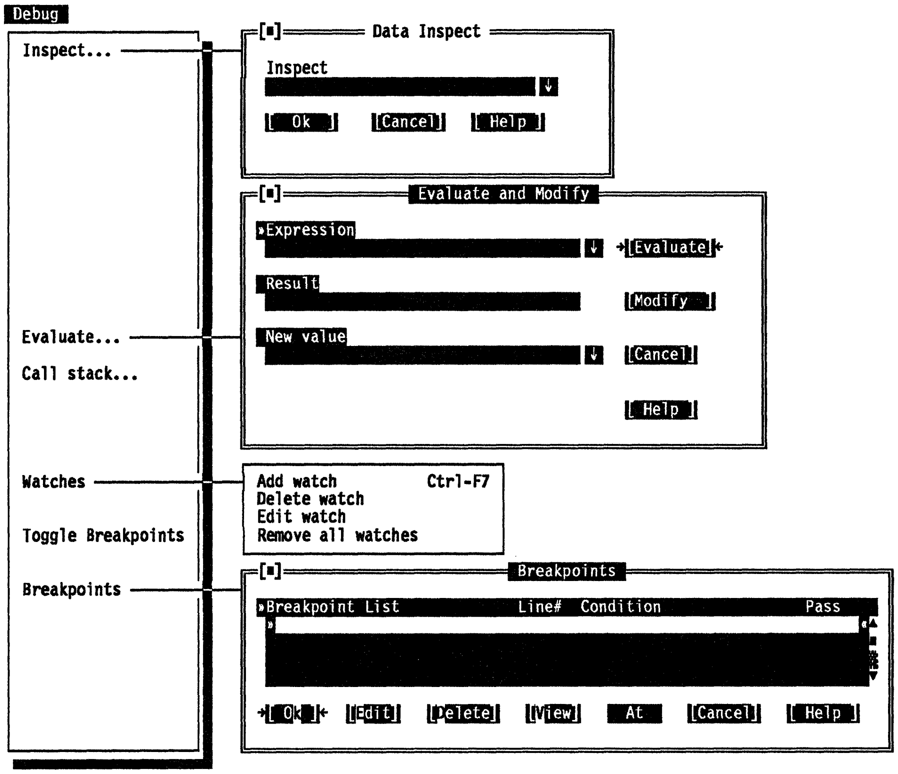
In Chapter 4, you learned the major elements of C. Through a variety of examples, you learned how these elements are put together to create working programs. With that knowledge, you've probably already written your own short programs. If not, now is a good time to begin, because there is nothing like writing your own code to test and extend your understanding of the concepts we have been discussing.

Of course, writing programs means dealing with your mistakes. Most experienced programmers agree that tracking down logical problems, commonly called *bugs*, in programs is a major part of program development. Debugging, or finding and fixing bugs, can often take longer than writing the program itself.

Turbo C++ comes with an integrated, source-level debugger that provides many capabilities of a standalone debugger, while giving you the convenience and speed of remaining within the integrated environment.

*Source level* means that you can trace through your actual program code—including, if you wish, every function that your code calls. By setting breakpoints, you can control how much of your program is run before you examine its condition. You can find out the current value of any variable by selecting it with the cursor or typing its name in the Evaluate field. You can set watches that monitor one or more variables and display their changing values as the program runs. All of this is done without your having to stop thinking in C (or C++), since you use the same variables, expressions, and operators you have been using to write your program.

Type TC to start Turbo C++ (if it isn't already running), and take a few minutes to examine the **Debug** menu. Highlight each item on the menu in turn, pressing *F1* each time to view the associated help text. These menu options are the debugging tools that you'll learn to use while developing this chapter's example program.



## Debugging and program development

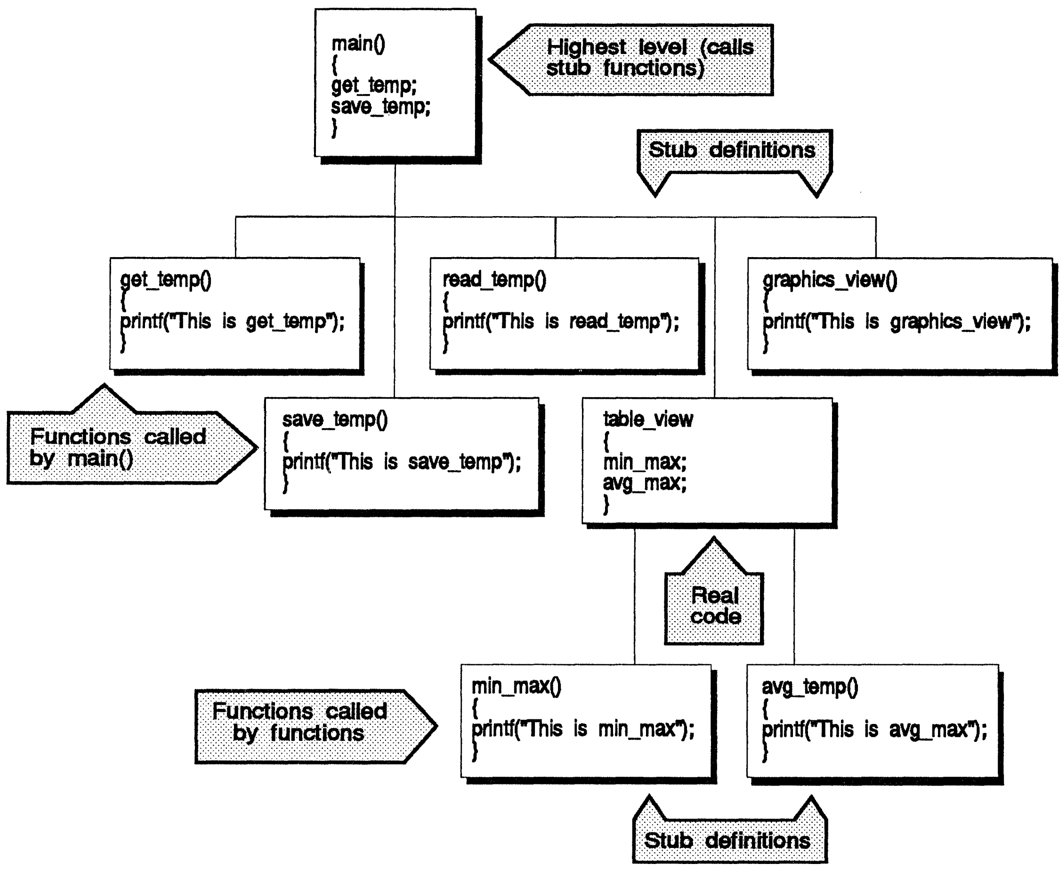
While this chapter focuses on debugging, it's important not to treat debugging as something separate from designing and writing a program. Turbo C++'s integrated debugger makes the mechanics of finding and fixing bugs as easy as possible, but the way you design your program can help make debugging easier, too.

Consider the old tube televisions. They were hard to repair because all their parts were wired into one big mass of tubes, resistors, capacitors, and so on. This often made it hard to find out which part or parts were not working. When you found the part that needed to be replaced, it was hard to get at that one part in the mass of wires.

After transistors and integrated circuits came into use, people started designing TVs differently. Each circuit came on its own slide-in module, which could slide out for easy access and testing. To replace a defective module, you simply slid in a replacement.

The C equivalent of these plug-in modules is the *function*. You'll design, implement, and test the example program in this chapter, one function at a time. It is often tempting to write a whole program, particularly if it is a small one, and only then start debugging. Like the old TV, though, this makes things unnecessarily difficult. To see why, suppose **main** calls a function **b**, which in turn calls functions **c** and **d**. If you don't test each of these functions as you develop them, it will be hard to tell whether a possible problem in function **d** is due to a coding error there, or to one in **b** or **c**. Of course, there is often more than just one bug. Meanwhile, a seemingly unconnected function **a** may have modified global data that **b** needs to pass to **c**. You can see that incremental program development—developing and testing one part of your program at a time—can save you considerable time and frustration.

Figure 7.1: Program development flowchart


main()
{
get_temp;
save_temp;
}
Highest level (calls stub functions)
Stub definitions
get_temp()
{
printf("This is get_temp");
}
read_temp()
{
printf("This is read_temp");
}
graphics_view()
{
printf("This is graphics_view");
}
Functions called by main()
save_temp()
{
printf("This is save_temp");
}
table_view
{
min_max;
avg_max;
}
Real code
Functions called by functions
min_max()
{
printf("This is min_max");
}
avg_temp()
{
printf("This is avg_max");
}
Stub definitions

## Designing the example program: PLOTEMP.C

The example program for this chapter collects temperature readings and displays them using either a table or graph view. It illustrates a diverse assortment of Turbo C++ capabilities: console and file I/O, passing an array to a function, and some graphics charting functions. Later, if you want, you can modify PLOTEMP.C to work with other kinds of data, and to provide different kinds of reports and charts.

One of the best ways to design a program is to prototype it by showing how it interacts with the user—how it requests information, responds to commands, and displays information. Before looking at the program code, let's see what the program will look like from the user's point of view.

When you run PLOTEMP.C, it displays this menu:

```
Temperature Plotting Program Menu
        E - Enter temperatures for scratchpad
        S - Store scratchpad to disk
        R - Read disk file to scratchpad
        T - Table view of current data
        G - Graph view of current data
        X - Exit the program
Press one of the above keys:
```

If you press *E*, you are prompted for a set of temperature readings. The program is set up to handle a set of eight readings, but you can change this simply by changing the line

```
#define READINGS 8
```

near the beginning of the program code.

Data entry will look something like this:

```
Enter temperatures, one at a time.
Enter reading # 1: 52
Enter reading # 2: 55
Enter reading # 3: 62
Enter reading # 4: 65
Enter reading # 5: 73
Enter reading # 6: 76
Enter reading # 7: 68
Enter reading # 8: 61
```

After this or any other menu selection except *X* (for E**x**it), the menu is redisplayed.

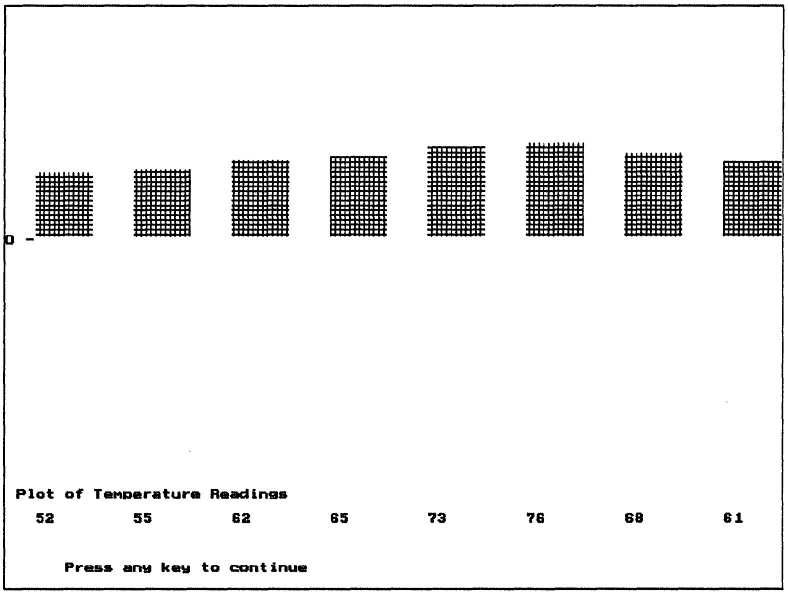
Choosing *S* saves the data set currently in memory to a disk file (you are prompted for the file name). Choosing *R* reads a data set from the disk file you specify to the program's "scratchpad" data array.

Once you have data in the scratchpad (either by entering it from the keyboard or reading it from disk), you can display a summary table of the data by choosing *T* (for Table view):

```
Reading            Temperature (F)
1                       52
2                       55
3                       62
4                       65
5                       73
6                       76
7                       68
8                       61
Minimum temperature: 52
Maximum temperature: 76
Average temperature: 64.000000
```

As an alternative, you can select a graph view of the current data by choosing *G*, as shown in the following figure:

Figure 7.2
Graph view of temperature data



```
Plot of Temperature Readings
  52          55          62          65          73          76          68          61

      Press any key to continue
```

There's one problem: The program you'll be putting together won't run entirely as expected. You'll have to use the debugger to find and fix the bugs.

## Writing the prototype program

Having decided what the program should do, you can determine what global data and other definitions the program will need, and write the **main** function:

*You can load this file right now: **File | Open** | PLOTEMP1. As you load and run these successive pieces of code, remember that we've deliberately sprinkled them with bugs.*

```
/* PLOTEMP1.C--Example from Chapter 7 of Getting Started */

/* This program creates a table and a bar chart plot from a
   set of temperature readings */

#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

/* Prototypes */

void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
void read_temps(void);

/* Global defines */

#define TRUE      1
#define READINGS  8

/* Global data structures */

int temps[READINGS];

int main(void)
{
   while (TRUE)
   {
      printf("\nTemperature Plotting Program Menu\n");
      printf("\tE - Enter temperatures for scratchpad\n");
      printf("\tS - Store scratchpad to disk\n");
      printf("\tR - Read disk file to scratchpad\n");
      printf("\tT - Table view of current data\n");
      printf("\tG - Graph view of current data\n");
      printf("\tX - Exit the program\n");
      printf("\nPress one of the above keys: ")

      switch   (toupper(getche()))
      {
         case 'E': get_temps();  break;
         case 'S': save_temps(); break;
         case 'R': read_temps(); break;
         case 'T': table_view(); break;
         case 'G': graph_view();
         case 'X': exit(0);
      }
   }
}

/* Function definitions */

void get_temps(void)
{
   printf("\nExecuting get_temps().\n");
}

void table_view(void)
{
   printf("\nExecuting table_view().\n");
}

void min_max(void)
{
   printf("\nExecuting min_max().\n");
}

void avg_temp(void)
{
   printf("\nExecuting avg_temp().\n");
}

void graph_view(void)
{
   printf("\nExecuting graph_view().\n");
}

void save_temps(void)
{
   printf("\nExecuting save_temps().\n");
}

void read_temps(void)
{
   printf("\nExecuting read_temps().\n");
}
```

Notice the following important features of this prototype program:

- Global #**define**s and data structures (the array *temps*).
- The function **main** provides the top-level menu.
- Other functions are declared with **void** return and argument type.
- Each function contains only a **printf** statement that identifies it when it is executing.
- The program is already complete enough to execute.

Why are functions given **void** declarations? They let you run the program and check the flow of execution at the top level. If you gave the functions full ANSI prototypes with arguments of various data types, you would have to start implementing the function definitions and writing code to use the values passed to the functions. Otherwise, you would receive compiler warnings about unused parameters. A good rule of program development is "do one step at a time, test one step at a time." At this point, you want to verify that the top-level structure of the program is sound.

Compile PLOTEMP1.C by choosing **C**ompile | **B**uild All. The compiler halts after displaying the error message:

```
Error C:\TC\EXAMPLES\PLOTEMP1.C 43:
   Statement missing ; in function   main
```

Turbo C++ has found a syntax error. You will usually have to wade through a flock of syntax errors before you can actually run your program, let alone debug it. Fortunately, syntax errors are usually easy to fix. Press the space bar: Here, the error bar highlighted the first line of the **switch** statement that displays the menu. A reference to a missing semicolon usually refers to the preceding statement (in this case, the last in the preceding group of **printf** statements). As you probably recall, you can press *Enter* or *F6* to switch to the Edit window and make your correction. If you have more than one syntax error to deal with, press *F6* to switch back to the Message window, and then use *Alt-F8* (or the ↓ key) to move to the next error listed.

Now that you've fixed the syntax error, compile the program again. This time the compiler and linker run without error messages, which means the program is now free of syntax errors.

Now choose **R**un | **R**un to run PLOTEMP1.EXE. You'll see the program's menu. Try pressing each of the keys listed in the menu—exercise the program. Try both uppercase and lowercase letters. Try letters that aren't given on the menu at all. What happens in each case? Did everything work as it should?

You probably noticed that with each selection made from the menu, a line describing the function was displayed. (Remember those **printf** statements in the stub functions in the listing?) Usually, the menu was then redisplayed. If you press *X* (or *x*), the program exits, and you are returned to Turbo C++. What happens when you press *G* (or *g*), though? You are also returned to Turbo C++. That's not right—you should get the menu again. There's a bug lurking somewhere.

## Tracing the flow of a program

By using options on the **Run** menu and observing the run bar, you can observe the order of execution of your program's statements and control how detailed the tracing will be.

Choose **Run** | **Trace** Into (or press *F7*). The debugger scrolls the beginning of function **main** into the Edit window and highlights it. This highlight is called the *run bar* and marks the *execution position*, indicating the next statement to be executed.

### Tracing high-level execution

To trace the high-level flow of your program, choose **Run** | **Step** Over (or press *F8*), and the next line containing code is executed (comment lines are skipped over). As you continue pressing *F8*, the run bar moves through the series of **printf** statements that display the menu. The screen appears to flicker. This is because each time the debugger executes a statement that displays information onscreen or executes a function call, it momentarily switches to the User screen. This is the screen display your program would generate if you weren't executing it within Turbo C++'s integrated environment (but the screen switching happens too fast for you to see the output). To look at the User screen, choose **W**indow | **U**ser Screen, or press *Alt-F5*. Depending on how far you've gotten in executing the **printf** statements, you'll see part or all of PLOTEMP's menu. Press any key to switch back to the debugger.

Continue pressing *F8* until you reach the first line of the **switch** statement. This line contains a call to the **getche** function, which requires the user to input a character. Any time user input is requested by the program, Turbo C++ switches to the User screen. Since you want to observe what happens when the graph view (G) option is selected, press *F8* again, then press *G*. After you supply the requested input, the display switches back to the debugger, and the run bar moves to the following statement:

```
case 'G': graph_view();
```

So far, so good. The next time you step, however, the following statement is `exit`, and you will be back in the editor the next time you step. Press *F8* to verify that the program finishes. By now, you've probably noticed that a **break** statement is needed on the preceding line. Correct the line so that it reads:

```
case 'G': graph_view(); break;
```

Now let's see if the fix worked. You could start the program and step one line at a time all the way from the beginning, but that would be tedious. Instead, choose **Run | Go** to Cursor (or press *F4*; make sure the cursor is on the correct line). Rebuild the program (say yes to the "OK to rebuild" prompt); its lines execute until user input is needed. Press *G* in response to PLOTEMP's menu; execution continues up to the line you just fixed. Continue stepping. Notice that this time the **break** statement is executed, and the execution position then goes back to the top of the **while** loop. PLOTEMP's menu now appears to be operating correctly.

### Tracing into called functions

When you use **Run | Step** Over, only the highest level of your program is stepped through. As you saw, the run bar stayed within the **main** function, stepping through the **printf** statements and the various **case**s within the **switch** statement. Often, however, you need to trace into the functions called by your **main** function, and sometimes the functions called by the functions. To trace through function calls, choose **Run | T**race Into, or press *F7*.

Try this now: Trace through the program, and press *E* at the PLOTEMP menu. This time, the run bar goes to the appropriate **case** in the **switch** statement, and then goes into the definition of the **get_temps** function. After stepping through the function (right now, there's just a **printf** statement there), execution falls to the bottom of the big **while** loop in **main**, then returns to the top where the menu is redisplayed.

## Continuing program development

Through the rest of this chapter, you will be adding one function at a time to PLOTEMP and then testing it by running the program again. Instead of making you do all the work of typing in code, we've provided incremental versions of PLOTEMP that you can load to complete each step. If you were debugging one of your own programs, you would follow these steps:

1. If necessary, replace the function prototype with the complete one.

2. Replace the stub function definition with the actual code needed to perform the task.
3. Test the new function by running the program again, making the appropriate choice from its menu.
4. Fix any bugs that crop up, testing until there are no more bugs.
5. Implement the next function in the same way, until the fleshed-out program is complete.

For best and fastest results, write, compile, run, and debug each function separately. Don't start developing the next function until you have a working program that has no apparent errors. This strategy won't eliminate all bugs, because "hidden" bugs sometimes continue to lurk, waiting for some unforeseen combination of circumstances. But this incremental approach minimizes the chance of unexpected crashes.

Start with the **get_temps** function, which gets a set of temperature readings from the keyboard. Since it doesn't take any arguments, and doesn't return anything directly, the current prototype declaration,

```
void get_temps(void)
```

doesn't need to be changed. (A full production version of this program would probably have this and every function return a value, so any errors could be signaled. We omit this here in the interest of keeping the size and complexity of the program manageable.)

Find the existing definition of **get_temps**. Right now, it's a stub definition that just prints a message when it executes. Replace the existing code with the following:

*For code that includes these changes, then load PLOTEMP2.C.*

```
void get_temps(void)
{
   char inbuf[130];
   int  reading;

   printf("\nEnter temperatures, one at a time.\n");
   for (reading = 0; reading < READINGS; reading++)
   {
      printf("\nEnter reading # %d: ", reading + 1);
      gets(inbuf);
      sscanf(inbuf, "%d", temps[reading]);

      /* Show what was read */
      printf("\nRead temps[%d] = %d", reading, temps[reading]);
      }
   }
```

## Setting breakpoints

*Remember that arrays in C begin with element 0: The first reading goes into element 0, the second into element 1, and so on.*

If **get_temps** works correctly, the **for** loop will prompt for each reading, then will use **gets** to get the reading as a string. Next, it uses **sscanf** to store it in the corresponding element of the *temps* array, and use another **printf** statement to display the value stored in the array. This second **printf** statement is just a temporary expedient: It lets you see how your inputs are being stored. After this function is debugged, you can remove it.

Run PLOTEMP2 and press *E* at the PLOTEMP menu so that **get_temps** executes. As you enter the data, something like this happens:

```
Enter reading # 1: 40

Read temps[0] = 0
Enter reading # 2: 50

Read temps[1] = 0
Enter reading # 3: 55

Read temps[2] = 0
Enter reading # 4: 57

Read temps[3] = 0
Enter reading # 5: 61

Read temps[4] = 0
Enter reading # 6: 64

Read temps[5] = 0
Enter reading # 7: 65

Read temps[6] = 0
Enter reading # 8: 60

Read temps[7] = 0
```

What's happening here? Regardless of what data you enter, the corresponding element of the *temps* array remains set to 0. Somehow the data being entered isn't finding its way into the array.

*A breakpoint is a place in your program where you want execution to run to, then stop at.*

Clearly, this code needs closer examination. To run the program until a particular area of interest is reached, set a breakpoint. Move the cursor to the beginning of the loop in the **get_temps** function:

```
for (reading = 0; reading < READINGS; reading++)
```

*If you choose Toggle Breakpoint for a line that already has a breakpoint, it removes that breakpoint.*

Now choose **D**ebug | **T**oggle Breakpoint (or press *Ctrl-F8*) to set a breakpoint at this line. The line with **for** is highlighted. Any time the flow of execution reaches a line at which you have set a breakpoint, the program is interrupted, and you're returned to the debugger. As you'll soon see, this lets you use other debugger commands to examine and change variables and other data structures. When you have several breakpoints in a program, you can choose **D**ebug | **B**reakpoints and select the View button in the dialog box to see the next breakpoint.

Breakpoints stay set until you do one of the following:

- Leave the integrated environment.
- Toggle the breakpoint off with *Ctrl-F8*.
- Delete the breakpoint using **D**ebug | **B**reakpoints | **D**elete.
- Delete the line(s) on which the breakpoints are set.
- Edit a file containing breakpoints and abandon the file without saving it.

Any time you correct a bug or otherwise edit the current file, and resume using debugger commands, Turbo C++ asks, "Source modified, rebuild?" Normally you'd press *Y* at this prompt, so the program can be rebuilt into a version that reflects your changes. If you press *N*, both breakpoints and the run bar will appear in the wrong places because the source file no longer matches the executable program.

## Instant breaking with *Ctrl-Break*

You probably know that pressing *Ctrl-Break* allows you to "break out" of many running programs. And so it is with Turbo C++ and the integrated debugger. However, the debugger doesn't always stop the program instantly. The debugger waits until the machine code corresponding to one of your lines of source code is being executed. It then stops the program at the machine instruction that corresponds to the beginning of the next source code line. The run bar appears on the line following the last line executed.

If you really want an instant break, press *Ctrl-Break* twice. When the second keypress is detected, the debugger terminates the program immediately, without flushing any output or calling any exit functions. (This is similar to using the **_exit** function.) This "double break" is usually undesirable, since the contents of data files become unpredictable, and the debugger no longer knows what line will be executed next. You'll probably only want to use a second *Ctrl-Break* when your program is "hung" or stuck in an infinite loop.

Run the program again; the PLOTEMP menu is displayed. Press *E*. The program runs into the **get_temps** function until the breakpoint is reached. Step with *F8* until the run bar has moved through the statements in the body of the **for** loop and returned to the first line of the loop. Now it's time to look at the key variables to learn more about the bug.

## Inspecting your data

Complex programs usually use a variety of data structures—arrays, structures, unions, lists, and so on. To understand what is going on in a particular part of your program, you often need to know the actual contents of these data structures. Turbo C++ provides a new facility called *inspectors*. Inspectors let you raise the hood at any part of your program and examine the inner workings.

### Inspector windows

*You can inspect any legal C or C++ expression, provided it doesn't contain symbols that were created with **#define** or function calls.*

To open an Inspector window for an item, move the cursor to the item in the Edit window and press *Alt-F4*. (You can also choose **D**ebug | **I**nspect.) Try inspecting the variable *reading* from your current version of PLOTEMP2.C (which should currently be in the Edit window). Choose **R**un | **T**race Into, then **D**ebug | **I**nspect.

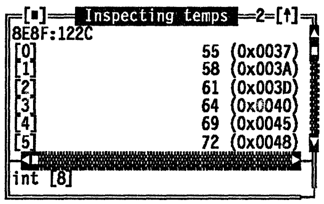
All Inspector windows begin with the name of the item. For variables, the line below contains the address of the variable, expressed as *segment:offset*. (Variables that have been declared to be of **register** type, or that have been optimized by Turbo C++ and thus placed in registers, don't have an address, since they are being stored in the CPU rather than in RAM. Instead, you'll see the word *register*.) The next line describes the item's data type (for example, **unsigned int**).

The actual value of the item is displayed to the right of the data type. Turbo C++ automatically selects the appropriate formats for displaying the type of data involved. For nonprinting characters, a backslash (\) followed by the hexadecimal character value is used in place of the character value. An **int** variable, on the other hand, would have decimal and hexadecimal values but no character representation. The other numeric types are handled similarly.

## Inspecting arrays and strings

For an array, a separate line is shown in the window for each element—if there are more elements than will fit in the window, use the arrow keys to scroll through them. For a string, a character representation for the string as a whole is also shown. The next figure shows the array *temps* from PLOTEMP.C after it has been filled with data via the **get_temps** function.

Figure 7.3
Inspecting the *temps* array



```
[■]═══ Inspecting temps ═2=[↑]
8E8F:122C
[0]                 55 (0x0037)
[1]                 58 (0x003A)
[2]                 61 (0x003D)
[3]                 64 (0x0040)
[4]                 69 (0x0045)
[5]                 72 (0x0048)
int [8]
```

The display for strings is similar to that for arrays (they are, after all, the same data structure). With strings, however, the character representation of the string is shown in addition to the individual elements.

## Inspecting structs and unions

For structures and unions, the values of the individual members are shown. To see how this works, load SOLAR.C (which you saw in Chapter 4, "An introduction to C") and inspect the array *solar_system*. Since this is an array whose elements are structures of type **planet**, you can browse through the array elements and view the data stored in the members of each **planet** structure.

## Inspecting pointers

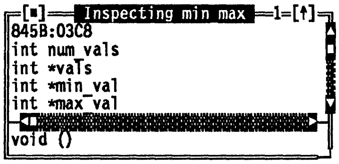
For a pointer, the Inspector window shows the address of the pointer, the address the pointer points to, and the data found at that address (which can be a simple variable, an array, a structure, and so on). Indexes ([0], [1], and so on) are shown to make it easy for you identify the position of each piece of data. This program defines a character pointer *ptr*:

```
main()
{
   char * ptr = "This is a string\n";
```

}

## Inspecting functions

The Inspector window for a function shows its return type and address, as well as all of the function's parameters. The Inspector window for the function **min_max** from the complete version of PLOTEMP.C is shown in the next figure.



```
845B:03C8
int num vals
int *vaTs
int *min_val
int *max¯val
void ()
```

Figure 7.4
Inspecting the min_max function

You can use function inspectors to review a function's return type and parameters without having to go back to the function declaration.

## When should you use inspectors?

Using inspectors may seem like overkill when dealing with simple variables and even arrays, since the **D**ebug | **E**valuate option (discussed later) also shows the values of variables. The real advantage of inspectors comes when dealing with more complex data structures (structures, unions, and arrays of these data types). In general, inspectors are most useful for studying your data in depth, while the **D**ebug | **E**valuate facility is best for taking a quick look at simple data.

## Evaluating and changing variables

Load PLOTEMP2.C, compile it, and step to the last **printf** in the function **get_temps**. Select *E* and enter one reading. You should now have executed the contents of the **for** loop once, and input, formatted, and stored one reading (via the **gets** and **sscanf** functions). Now choose **D**ebug | **E**valuate, which pops up a window containing three fields:

- the Expression field, which contains the expression you are interested in
- the Result field, which displays the value of the expression in the Evaluate field

- the New Value field, where you can enter a new value for the selected expression

By default, the "word" (C variable, keyword, function call, and so on) at the cursor is displayed in the Evaluate field. You're interested in two variables: *reading*, which is the loop's counter variable, and the array *temps*, which is supposed to contain the input data.

Type `reading` and press *Enter*. The debugger now displays `0` in the Result field. If you step through the statements in the loop and evaluate *reading* again, you'll find its value is now 1.

You can also examine the values of more complex data structures, such as arrays, strings, and structures. Here, you want to know more about what's happening to the array *temps*. Go ahead and type `temps` in the Evaluate field. The following is displayed:

```
{0, 0, 0, 0, 0, 0, 0, 0}
```

This shows the integer values currently stored in the array *temps*. You can get a single value by using an index: Specifying `temps[0]` would get you the first integer, for example. Or you could open an Inspector window.

Notice that we have been referring to expressions, not just values (such as variables) by themselves. Recall that an expression is a combination of variables, constants, and operators that yields a single value; for example, `vals1[index] + vals2[index] + 1`. You can display the value of any expression, provided that

- It doesn't involve a function call (so an expression such as `sqrt(a) + 1` can't be used).
- It doesn't use a **#define** value, such as READINGS in the current program.

For practice, get the values of the following expressions by typing them into the Evaluate field:

```
reading + 2
temps[reading + 1]
```

## Specifying display format

Optionally, you can add a comma and a format specifier to the value you want displayed. For example, type `reading,h` to see the current value of *reading* in hexadecimal (you could also type `reading, x`). By default, integers are displayed as decimal, and character arrays are displayed as strings.

The specifier **m** is useful when dealing with arrays: It displays a memory dump starting at the specified address. For example, `temps,m` gets you a memory dump starting at the location specified (since *temps* is an array, its name points to the starting address of stored data):

```
00 00 00 00 00 00 00
```

This shows that all the elements of the array *temps* are currently set to 0. The number of elements displayed depends on the size of the array. You can combine **m** with other specifiers:

```
temps,mh
```

displays a memory dump in hexadecimal format.

Another useful specification is **p**, which displays the selected variable as a pointer, giving information about the area of memory being pointed to (for example, the Interrupt Vector Table, the BIOS data area, or the user program's program segment prefix [PSP]). If the memory pointed to is within the program's own allocated memory, the name of the variable (if any) at the address of the segment offset is also displayed. (Chapter 4, "Memory models, floating point, and overlays," in the *Programmer's Guide* has more information about the regions of memory involved.)

## Specifying the number of values

When dealing with an array, you can specify how many values you want to display: `temps,5` specifies the first five elements of the array *temps*. You can combine this repeat count with format specifiers. For example, `temps[2],3h` specifies that the three elements following the third element of the *temps* array should be displayed in hexadecimal format.

Other format specifiers and more details of the debugger commands are given in the *User's Guide*, Chapter 1, "The IDE reference" (a reference chapter for the integrated environment). After you practice using the debugger in this chapter, we recommend reading the pertinent portions of that chapter.

## Copying from the cursor position

As noted earlier, the Evaluate field contains whatever word was at the cursor position when you chose **Debug | Evaluate**. You can take advantage of this to save typing. For example, if you move the cursor to the beginning of the expression

```
temps[reading]
```

*temps* appears in the Evaluate field. As you press →, the characters following the word *temps* appear. You can thus copy the complete expression `temps[reading]` into the Evaluate field, then press *Enter* as usual to display the value.

## Specifying variables in other functions

Right now, you're looking at the variables *reading* and *temps* in the function **get_temps**. You can also ask the debugger for the values of static variables in other functions because static variables retain their values even when the function that uses them isn't being executed. You can also look at variables in the functions that called the one you're executing. You can't look at ordinary automatic variables declared in other functions because they no longer have a value when their function exits. The context in which expressions are evaluated is given by the current cursor position in the Edit window.

To specify a variable outside of the current function, you can move the cursor into the body of the function, or you can give the name of the function, a period, and the name of the variable.

If the variable you want to inspect is in another program module, you must specify the module name first; for example,

```
module2.getvals.count
```

## Changing values

Now you know how to look at different kinds of variables and expressions, and how to see their values in different formats. Practice stepping through the **for** loop in **get_temps** and examine the values of *reading* and *temps[reading]*. The latter insists on always being 0.

Try evaluating the expression `temps[reading]` as you step a few times through the loop. The debugger displays its value as 0, regardless of the value of *reading*. But what should this expression represent? Since it is specifying the storage destination for the **sscanf** function, it needs to be an address. This means that all of the values entered are being stored at address 0! You can confirm this by using the pointer format `temps[reading],p` and finding that that value is still 0.

You need to use the address operator **&** to make this expression refer to the address of the *temps* array. Evaluate `&temps[reading],p`. The result will look something like this: `DS:1278 temps+1`. The actual values displayed will depend on your system configuration and the current value of *reading*, but you can see that `&temps[reading]` points to a bona fide address in the data segment, with an offset from the address pointed to by the variable *temps*.

Change the expression `temps[reading]` in the **sscanf** statement to `&temps[reading]`. If you now continue stepping through the program, you will be asked whether to rebuild the program; press *Y* for *Yes*. Now, if you step through the **for** loop again and evaluate `temps[reading]`, you'll find that the values you are entering are stored correctly in the array.

This is a good time to practice changing values with the debugger: Evaluate the current value of `temps[reading]`.

Use the *Tab* key to move among the three fields (Evaluate, Result, and New Value). Once the input cursor is in the New Value field, type a value such as `66`, and press *Enter*. Now, if you type `temps[reading]` in the Evaluate field, its new value, 66, is shown in the Result field. You can change the value of any expression that represents a single data element, such as a simple variable, a pointer, or an array element.

Changing values interactively with the debugger is useful for temporarily fixing a bug and continuing program execution a little further, looking for the next bug. Here, for example, you could put new values in *temps*[*0*] through *temps*[*7*], set *reading* to 8 to break out of the **for** loop, and return to the program's main menu. You can also force a function to return a specific value, or to pass that value to another function. This lets you test unusual conditions that might lead to bugs, without having to put temporary assignment statements in your code.

The **get_temps** function should now be working correctly. Next, implement the **table_view** function, so you can view the entered data. Replace the stub code for **table_view** with the following:

*Load in PLOTEMP3.C.*

```
void table_view (void)
{
   int reading;

   clrscr();         /*clear the screen */
   printf("Reading\t\tTemperature (F)\n");

   for (reading = 0; reading <= READINGS; reading++)
      printf("%d\t\t\t%d\n", reading + 1, temps[reading]);

   min_max();
   printf("Minimum temperature: \n");
   printf("Maximum temperature: \n");
   avg_temp();
   printf("Average temperature: \n");
}
```

This function prints the table headings and then uses a **for** loop to obtain and print the values stored in the *temps* array. Eventually, it will also print the minimum, maximum, and average temperatures. These functions haven't been implemented yet, so they'll just print out a message saying that they are being executed.

Notice that including the called functions before they are implemented lets you test the program flow, and reminds you of the program structure. This kind of design is often called *top-down* because the program is designed at the top level first (the **main** function). You are in the process of designing the functions called directly by **main**, such as **table_view** here. When the top level of **table_view** is working, you'll then implement the functions it calls—**min_max** and **avg_temp**.

Now build and run PLOTEMP3.C, choose *E*, and enter test data (we entered 10, 20, 30, 40, 50, 60, 70, and 80). When you are back at the menu, choose *T* (Table view), and you'll see something like this:

```
Reading           Temperature (F)
   1                     10
   2                     20
   3                     30
   4                     40
   5                     50
   6                     60
   7                     70
   8                     80
   9                      0

Executing min_max().
Minimum temperature:
Maximum temperature:

Executing avg_temp().
Average temperature:
```

## Monitoring your program by setting watches

Well, you entered eight readings and got back nine! The last one had a value of 0. You probably suspect the infamous "one off" bug—getting one less or one more iteration of a loop than you had expected.

First localize the problem by setting a breakpoint at the first line of the **for** loop in **table_view**. To do this, move the cursor to that line and choose **D**ebug | **T**oggle Breakpoint (or press *Ctrl-F8*).

Now run the program again, enter the test data, and choose Table view. The program stops at the breakpoint in **table_view**; now you can see what's going on in this **for** loop.

So far, you have obtained the values of variables by stepping through the program and using **D**ebug | **E**valuate to inspect their values. This is fine when you just need to inspect the values once, but when you're dealing with loops or repeated function calls, you also want to see how the values change. It would be very tedious to evaluate the variables by hand repeatedly. The debugger lets you monitor these changing values automatically by setting watches. A *watch* is an expression whose value is updated each time it is encountered in the running program.

### Adding a watch

You are interested in two variables in this case: *reading*, which is incremented repeatedly by the **for** loop, and *temps[reading]*, which holds the value being printed each time through the loop. Since the cursor is already nearby, the easiest way to set these watches is to move the cursor to the name of the variable you want to watch, and then choose **D**ebug | **W**atches | **A**dd Watch (or press *Ctrl-F7*). Move the cursor to *reading* and try this; you'll see the pop-up window. As with **D**ebug | **E**valuate, the default name shown is the one at the cursor; simply press *Enter*. Use the same procedure to set a watch for *temps[reading]*. The pop-up window shows `temps`, but as with **D**ebug | **E**valuate, you can use → to copy the rest of the expression into the window, and then press *Enter* to set the watch.

## Watching your watches

Now that you have set two watches, the variable name and value for each watch is shown in the Watch window:

```
reading: 177
temps[reading]: 92
```

Since the loop hasn't been run yet, the values shown (which may be different on your system) are meaningless, representing whatever happens to be at the respective memory locations.

Now start stepping through the loop (with *F8*). As you step, notice how the values change. After the first time through the loop, the values are

```
reading: 0
temps[reading]: 10
```

assuming you entered your test data starting with 10 as described earlier. The next time through the loop, the watches display

```
reading: 1
temps[reading]: 20
```

The last time the loop is executed, the values are

```
reading: 8
temps[reading]: 0
```

This suggests that the loop exits only after *reading* reaches 8. When should it exit? Since there are eight readings entered, and *reading* starts at 0, the last value it should have during the loop is 7, not 8. Now take a look at the loop exit condition:

```
reading <= READINGS
```

Checking your #**define**s at the beginning of the program, you see that READINGS is 8. Do you see the problem? To exit when *reading* is 7 (after processing eight readings), the condition should read `reading < READINGS`. Fix this, and rerun the program to see if it works correctly.

## Controlling the debugger windows

If you have set more than a few watches, there won't be room to see them all at once. You can scroll the Watch window using the

*PgUp* and *PgDn* keys, or move one line at a time with the ↑ and ↓ keys.

*This also works within the **D**ebug | **E**valuate window.*

If a particular watch expression is too long to fit in the window, you can see its beginning and end by scrolling it with the *Home, End,* ←, and → keys.

Another way to see more is to zoom a window. See Chapter 1, "The IDE reference," in the *User's Guide* for how to handle the new environment's windows.

Remember that you can look at an entire screen of the program output at any time by pressing *Alt-F5*. Press any key to return to the environment.

Now is a good time to practice using these features.

## Editing and deleting watches

It's easy to edit, add, or delete watches. When the Watch window is active, the currently active expression is highlighted. To select a different expression, use the *Home, End,* ↑, or ↓ keys.

*You can't change the value of the expression, only the expression itself. To change the value, use **D**ebug | **E**valuate.*

To edit (change) the currently highlighted watch, you can choose **D**ebug | **W**atches | **E**dit Watch. Even easier, as shown on the bottom line of the screen, you can press *Enter*. The debugger opens a pop-up window with the selected expression, and you can edit it. Practice by changing the watch for `temps[reading]` to `temps[reading+1]`.

You already know how to add watches, but once the Watch window is active, there's an easier way: Press *Ins*. A pop-up window appears. You can type in the watch expression, add to it with the → key, or accept the default that was copied from the cursor position.

To delete the current watch, choose **D**ebug | **W**atches | **D**elete Watch, or simply press *Del*. Practice by deleting the watch for *reading*. You can delete all of the watches by choosing **D**ebug | **W**atches | **R**emove All Watches.

## Finding a function definition

Now that PLOTEMP.C is starting to flesh out, it's harder to find the function you want to examine. The debugger provides a way to scroll the Edit window to a specified function definition. Choose **S**earch | **L**ocate Function, and Turbo C++ opens a dialog box. Practice by typing `get_temps` (don't type the parentheses after the function name, or the debugger won't find your function). The definition of **get_temps** is now displayed in the Edit window. This is useful for reviewing the definition of a function, as well as for finding locations to set breakpoints and watches.

Note that **S**earch | **L**ocate Function works only with functions that have source code in a file that has been compiled with debug information. Library functions such as **printf** can't be found with **S**earch | **L**ocate Function, since their source code isn't available in the integrated environment.

## Finding out who called whom

In a complex program, there may be several levels of function calls, and you may find it hard to remember the order in which functions were called as the program executed to a particular breakpoint. The debugger can help you out here, too. Set a breakpoint that will halt the program at the place where you want to see the call sequence. For practice, set a breakpoint at the **printf** in **min_max**.

Run the program and choose Table View. This will cause the program to stop at the breakpoint. Now choose **D**ebug | **C**all Stack. A pop-up window lists all the functions that are waiting to finish execution at this point. The call stack has the most recently called function at the top, which in this case is **min_max**. It was called by **table_view**, which in turn was called by **main**.

*You can always scroll back to the current execution position by choosing the first function in the Call Stack window—in this case, **min_max**.*

You can use the ↑ and ↓ keys to highlight a particular function in the Call Stack window. If you press *Enter*, the Edit window scrolls to show the last line executed in that function. Right now,

- the last line executed in **min_max** is the first line of its definition, since that's where you placed the breakpoint.
- the last line executed in **table_view** was the line containing the call to **min_max**.
- the last line executed in **main** is the line that executed **table_view**; namely, `case 'T': table_view; break;`.

In other words, **min_max** is currently being executed, and **table_view** and **main** are pending completion.

### Multiple source files

As you work with longer programs, you'll find the features we've been discussing to be especially useful. Many substantial programming projects consist of several source files. The debugger automatically loads the file needed to fulfill your request into the Edit window. For example, if you use **S**earch | **L**ocate Function to find a function that is declared in a source file other than the one in the Edit window, the debugger loads the appropriate source file into the editor. If you've made any changes to the current file, you are first asked if you want to save the changed file to disk. The same thing happens when you use **D**ebug | **C**all Stack to examine the last executed line of a function whose definition is in a different source file.

Although the debugger makes it easy to work with multiple source files, it is good practice to debug only one or two source files at a time. Always test a given "bug fix" before moving on because there is always a chance that your fix did not work, or possibly even introduced new bugs.

## Preventive medicine

You'll soon resume the development and testing of PLOTEMP.C. To aid in this and future debugging efforts, take a look at some ways to minimize bugs, and look at some common "buggy" situations.

### Design defensively

Just as you can avoid accidents by driving your car defensively, you can avoid bugs by designing your program defensively. As you've seen, the design of PLOTEMP.C represents an approach to defensive design through top-down programming.

Try to build up your program from functions whose purposes are simple and well-defined. This makes it easier to set up test cases and analyze their results. It also makes your program easier to read and modify. For example, if PLOTEMP.C combined both table and graph views in the same function, the code could easily become unwieldy.

Try to minimize the number of data elements each function requires and the number of elements it changes. This too makes it easier to set up test cases and analyze their results, and to read and modify your program. It also tends to limit the amount of havoc a misbehaving function can cause, letting you run the function several times in a single debugging session. A program designed this way is said to be *loosely coupled.*

### Write clearly

Write your code cleanly, with consistent indentation, liberal comments, and descriptive variable names.

Keep your code simple. Express complicated operations in many simple statements rather than a few complex ones that show off your knowledge of C's more obscure features. Turbo C++'s code optimization makes your code reasonably efficient, and it will be much easier to debug, read, and modify.

Don't try to squeeze the last bit of efficiency out of your program when you write it. When you try to make code as efficient as it can be, it also tends to become hard to read and debug. If your program turns out to be too slow when it's done, that's the time to decide which parts are worth speeding up, and how best to do it. (Turbo Profiler is the best tool for that task, anyway.)

Be alert for opportunities to write functions that can be used more than one way in your program, or can be reused in other programs. Writing and debugging one generalized function is usually easier than writing two or more specialized ones.

## Systematic software testing

Before a jet liner takes off, the crew goes through a systematic checklist to ensure that everything is working properly. Following a specific routine reduces reliance on fallible human memory. In the same way, you should work toward a standard approach to software testing: A checklist of steps that your experience shows will lead you to a reliable program.

There is no one "right" way to test a program; your checklist will depend on the types of programs you write, your strengths and weaknesses as a programmer, and your personal style. The following checklist can serve as a starting point, since it reflects widespread experience.

- *Feed the program some input that is simple but not trivial.* Try the unusual—for example, have you tried entering negative temperatures into PLOTEMP? Trace into the code using **D**ebug | **E**valuate and watch expressions liberally to check the values of data items. Correct the bugs you find, one or a few at a time.
- *Feed the program other sets of data that let you exercise the parts you couldn't test in the preceding step.* If possible, have someone who is unfamiliar with your program interact with it at the keyboard. Common experience shows that programmers have difficulty exercising their own programs properly because they know which values are appropriate and which aren't. If your program is designed to be used by accountants, try to find an accountant.
- *Test every statement in your program.* You may find bugs where you didn't suspect they could exist.
- *Put aside the debugger and test the entire program for correct behavior.* If the program will be used by other people who will expect it to be well-behaved, test its response to every type of error it could possibly encounter. A program that handles most types of errors well is said to be robust.

## Test modifications thoroughly

When you modify a program, retest the affected parts thoroughly. You may have to retest parts that haven't changed but are affected by the changes.

If the program is complex, keep a record of the tests you have performed. When you modify the program, this record helps you repeat all the tests whose results could possibly be affected by the change. If the tests involve particular input files, save the files.

## Areas to watch carefully

As you continue to learn C and to develop programs, keep a list of common bugs and coding errors, and check it as part of your debugging session. Here are some areas where many programmers run into trouble:

- making out-of-bounds errors
- confusing addresses versus values at those addresses
- placing the increment and decrement operators incorrectly
- not testing statements thoroughly
- using Pascal syntax instead of C

Each of these is discussed next.

*Give special attention to boundary conditions—conditions that make a program escape from a loop, fill an array, and so on.* Bugs are especially likely to be manifested as failures to handle boundary conditions correctly. You've already seen how the condition `reading <= READINGS` caused one too many values to be displayed. Other problems could be caused by starting at 1 instead of 0.

*Always be careful about whether you are specifying an address or a value at that address.* For example, don't confuse the value *temps*[*reading*] with the address &*temps*[*reading*].

*Be careful with the increment and decrement operators* **++** *and* **– –**. Is the value being incremented before or after it is used?

*Be alert for individual statements or expressions that must be tested more than one way, like these:*

```
switch ( strcmp(a,b) ) ...
```

**strcmp** can return three values: 0 (*a* equals *b*), –1 (*a* is less than *b*), or +1 (*a* is greater than *b*). This suggests that you should test the statement with three sets of input values to verify that **strcmp** makes it do the right thing in each case.

```
x = (x>0) ? func(x) : 0 ;
```

This statement contains an "implicit if" that can produce two different results.

## Finishing PLOTEMP.C

You've installed and tested the prototype PLOTEMP.C and have implemented, tested, and debugged the **get_temps** and **table_view** functions. You've learned how to use all of the debugger features. The completion of PLOTEMP.C involves a series of exercises where you

- Replace the stub code for a particular function with the code given in the listing. We've made this easy for you by providing code for each step that has the corrections made to that point. All you need do is load the next version.
- Change the function prototype (if necessary).
- Test the function implementation using appropriate debugger facilities.
- Find and fix the bugs.
- Move on to the next function.

The answers are given at the end of this chapter.

## Finishing table_view

To finish this function, you need to implement the following two functions:

*Load PLOTEMP4.C. Remember that we have included deliberate bugs in the PLOTEMP programs so you can practice your debugging skills.*

```
void min_max (int num_vals, int vals[], int *min_val, int *max_val)
{
   int reading;

   *min_val = *max_val = vals[0];

   for (reading = 1; reading < num_vals; reading++)
   {
      if (vals[reading] < *min_val)
         *min_val = &vals[reading];
      else if (vals[reading] > *max_val)
         *max_val = &vals[reading];
   }
}

float avg_temp(int num_vals, int vals[])
{
   int reading, total = 1;

   for (reading = 0; reading < num_vals; reading++)
      total += vals[reading];

   return (float) total/reading; /* reading equals total vals */
}
```

Since these functions have parameters and return values, change their prototypes to

*This change is included in PLOTEMP4.C.*

```
void min_max (int num_vals, int vals[], int *min_val, int *max_val)

float avg_temp(int num_vals, int vals[])
```

Finally, change **table_view** so that the return values from the functions are used properly. The revised **table_view** should read as follows:

*This change is included in PLOTEMP4.C*

```
void table_view (void)
{
   int reading, min, max;

   clrscr();                         /* clear the screen */
   printf("Reading\t\tTemperature(F)\n");

   for (reading = 0; reading < READINGS; reading++)
      printf("%d\t\t\t%d\n", reading + 1, temps[reading]);

   min_max(READINGS, temps, &min, &max);
   printf("Minimum temperature: %d\n", min);
   printf("Maximum temperature: %d\n", max);
   printf("Average temperature: %f\n", avg_temp(READINGS, temps));
}
```

Now for some debugging on your own. Check normal operations:

✓ ■ Are the loops working properly?
✓ ■ Are the arithmetic operations appropriate?
✓ ■ What do the comparisons compare?

## Implementing graph_view

Recall that the **graph_view** function creates the chart shown in Figure 7.2. To implement this function, replace its definition with the following (and be sure to add `#include <graphics.h>` at the beginning of your code):

*Load PLOTEMP5.C.*

```
void  graph_view(void)
{
   int graphdriver = DETECT, graphmode;
   int reading, value;
   int maxx, maxy, left, top, right, bottom, width;
   int base;                 /* zero x-axis for graph */
   int vscale = 1.5;         /* value to scale vertical bar size */
   int space = 10;           /* spacing between bars */

   char fprint[20];          /* formatted text for sprintf */

   initgraph(&graphdriver, &graphmode, "");
   if (graphresult() < 0)        /* make sure initialized OK */
      return;

   maxx  = getmaxx();            /* farthest right you can go */
   width = maxx / (READINGS + 1); /* scale and allow for spacing */
   maxy  = getmaxy() - 100;      /* leave room for text */
   left  = 25;
   right = width;
   base  = maxy / 2;             /* allow for neg values below */

   for (reading = 0; reading <= READINGS; reading++)
   {
      value = (temps[READINGS]) * vscale;
      if (value > 0)
      {
         top = base - value;     /* toward top of screen */
         bottom = base;
         setfillstyle(HATCH_FILL, 1);
      }
      else
      {
         top = base;
         bottom = base - value;  /* toward bottom of screen */
         setfillstyle(WIDE_DOT_FILL, 2);
      }
      bar(left, top, right, bottom);
      left  += (width + space);  /* space over for next bar */
      right += (width + space);  /* right edge of next bar */
   }

   outtextxy(0, base, "0 -");
   outtextxy(10, maxy + 20, "Plot of Temperature Readings");
   for (reading = 0; reading < READINGS; reading++)
   {
      sprintf(fprint, "%d", temps[reading]);
      outtextxy((reading *(width + space)) + 25, maxy + 40, fprint);
   }

   outtextxy(50, maxy+80, "Press any key to continue");

   getch();                      /* wait for a key press */

   closegraph();
}
```

## save_temps and read_temps

The function **save_temps** saves the current "scratchpad" (the contents of the array *temps*) to a disk file. By now, you should be familiar with the logic involved in accessing elements of this array.

Replace the stub definition for the **save_temps** function with the following:

*Load PLOTEMP5.C.*

```
void  save_temps(void)
{
   FILE * outfile;
   char file_name[40];

   printf("\nSave to what filename? ");
   while (kbhit());  /* "eat" any char already in keyboard buffer */
   gets (file_name);
   if ((outfile = fopen(file_name,"wb")) == NULL)
      perror("\nOpen failed! ");
      return;
   fwrite(temps, sizeof(int), READINGS, outfile);
   fclose (outfile);
}
```

The function **read_temps** is the counterpart to **save_temps;** it reads values from a disk file into the *temps* array. Implement **read_temps** by replacing its stub definition with the following:

*This change is also included in PLOTEMP5.C.*

```
void read_temps(void)
{
   FILE * infile;
   char file_name[40] = "test";

   printf("\nRead from which file? ");
      gets(file_name);

   while (kbhit());   /* "eat" any char already in keyboard buffer */

   if((infile == fopen(file_name,"rb")) == NULL)

      perror("\nOpen failed! ");

   fread(temps, sizeof(int), READINGS, infile);
   fclose (infile);
}
```

After you're finished with **read_temps**, you should have a complete, working version of PLOTEMP.C. (PLOTEMP6.C is a bug-free version of this program.)

## Answers to debugging exercises

Here are the bugs in the remaining functions of PLOTEMP5.C.

### min_max and avg_temps

In **min_max**, the **if** statements assign the value *&vals*[*reading*] (which is an address) to the *min* or *max*, rather than the correct value *vals*[*reading*]. Also, in **avg_temp**, the variable *total* should be

0. Having it start as 1 adds 1 to the total of the readings, thereby giving an incorrect average.

By the way, note that this function receives and passes pointers to the calling function **table_view**—not the actual values of the variables. For more on this, review the material on pointers in Chapter 4.

## graph_view

There are two sneaky bugs in this function. They are in the **for** loop:

```
for (reading = 0; reading <= READINGS; reading++)
{
   value = temps[READINGS] * vscale;
```

The value being read is *temps*[READINGS]—the constant READINGS instead of the variable *reading*. The result is that the only value that will be graphed is the nonexistent element *temps*[8]. In the second bug, the condition `<= READINGS` should be `< READINGS` to read the correct number of values. Notice how the first bug masks the second—often you can't detect a given bug until you've fixed another bug, since the first bug prevents proper execution of the code containing the second bug.

## save_temps

Here, the problem is not in the first line of the **for** loop, but rather in the body:

```
if ((outfile = fopen(file_name,"wb")) == NULL)
   perror("\nOpen failed! ");
   return;
```

Braces are needed to place both the **perror** statement and the **return** statement under the jurisdiction of the **if**. Thus the function always returns at this point, even if the file was opened correctly.

The compiler warning "Unreachable code in function save_temps" means that the line following the return statement can never be executed. Without the braces, the return is always executed.

## read_temps

When you compiled this, you should have seen the compiler warnings "Possible use of infile before definition". Consider the following code:

```
if ((infile == fopen(file_name,"rb")) == NULL)
```

When you open a file, the file handle *infile* should be getting a value from **fopen**; the value is tested to see if it is NULL, indicating that the file opening failed. Why isn't *infile* being defined (getting a value) as it is first used? The reason is that **==**, rather than **=**, follows *infile*. Since **==** indicates a comparison rather than an assignment, *infile* isn't getting a value.

# Bibliography

Many leading book publishers support Borland products with a wide range of excellent books, serving everyone from beginning programmers to advanced users. Of course, since Turbo C++ is a new product, most of the Turbo C books in this bibliography are specific to Turbo C 2.0. They are, nonetheless, useful, and there are four books specific to Turbo C++.

## Beginning to intermediate

Burnap, Steve. *COMPUTE's Turbo C for Beginners*. Radnor, PA: COMPUTE! Publications, 1988.

Derman, Bonnie (editor) and Strawberry Software. *Complete Turbo C*. Glennview, IL: Scott, Foresman & Co, 1989.

Edmead, Mark. *Illustrated Turbo C*. Plano, TX: Wordware Publishing, 1989.

Goldstein, Larry and Larry Gritz. *Hands On Turbo C*. New York, NY: Brady Books, 1989.

Hergert, Douglas. *The ABC's of Turbo C 2.0*. Alameda, CA: Sybex, Inc, 1989.

Jamsa, Kris. *Turbo C Programmer's Library*. Berkeley, CA: Osborne/ McGraw-Hill, 1988.

Kelly-Bootle, Stan. *Mastering Turbo C*, 2nd edition. Alameda, CA: Sybex, Inc, 1989.

LaFore, Robert. *The Waite Group's Turbo C Programming for the PC*, revised edition. Indianapolis, IA: Howard W. Sams & Co, 1989.

Miller, Larry and Alex Quilici. *The Official Borland Turbo C Survival Guide*. New York, NY: John Wiley & Sons, 1989.

Pohl, Ira and Al Kelley. *A Book on C*. Menlo Park, CA: Benjamin/ Cummings, 1984.

Pohl, Ira and Al Kelly. *Turbo C by Dissection*. Menlo Park, CA: Benjamin/Cummings, 1987.

Pohl, Ira and Al Kelly. *Turbo C, The Essentials of Programming.* Menlo Park, CA: Benjamin/Cummings, 1988.

Schildt, Herbert. *Using Turbo C++.* Berkeley, CA: Osborne/ McGraw-Hill, 1990.

Voss, Greg and Paul Chui. *Turbo C++ DiskTutor.* Berkeley, CA: Osborne/McGraw-Hill, 1990.

Wiener, Richard. *Turbo C at Any Speed.* New York, NY: John Wiley & Sons, 1988.

Zimmerman, S. Scott and Beverly Zimmerman. *Programming with Turbo C.* Glennview, IL: Scott, Foresman & Co, 1989.

## Advanced

Alonso, Robert. *Turbo C DOS Utilities.* New York, NY: John Wiley and Sons, 1988.

Burnap, Steve. *COMPUTE's Advanced Turbo C Programming.* Radnor, PA: COMPUTE! Publications, 1988.

Davis, Stephen R. *Turbo C: The Art of Advanced Program Design, Optimization and Debugging.* Redwood City, CA: M & T Books, 1987.

Ezzel, Ben. *Graphics Programming in Turbo C.* Reading, MA: Addison-Wesley, 1989.

Goldentahl, Nathan. *Turbo C Programmer's Guide.* Chesterland, OH: Weber Systems, Inc, 1988.

Hunt, William. *The C Toolbox.* Reading, MA: Addison-Wesley, 1985.

Johnsonbaugh, Richard and Martin Kalin. *Applications Programming in Turbo C.* New York, NY: Macmillan Publishing Co, 1989.

Mosich, Donna, Namir Shammas, and Bryan Flamig. *Advanced Turbo C Programmer's Guide.* New York, NY: John Wiley & Sons, 1988.

Murray, Bill and Chris Pappas. *Turbo C++ Professional Handbook.* Berkeley, CA: Osborne/McGraw-Hill, 1990.

Porter, Kent. *Stretching Turbo C.* New York, NY: Brady Books, 1989.

Schildt, Herbert. *Advanced Turbo C*, 2nd edition. Berkeley, CA: Osborne/McGraw-Hill, 1989.

Stevens, Al. *Turbo C: Memory Resident Utilities, Screen I/O and Programming Techniques*. Portland, OR: MIS: Press, 1987.

Weiskamp, Keith. *Advanced Turbo C Programming*. Boston, MA: Academic Press, 1988.

Young, Michael. *Systems Programming in Turbo C*. Alameda, CA: Sybex, Inc, 1988.

## Object-oriented programming

Dewhurst, Stephen C. and Kathy T. Stark. *Programming in C++*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Eckel, Bruce. *Using C++*. Berkeley, CA: Osborne/Mcgraw-Hill, 1990.

Lippman, Stanley B. *C++ Primer*. Reading, MA: Addison-Wesley, 1989.

Pohl, Ira. *C++ For C Programmers*. Menlo Park, CA: Benjamin/Cummings, 1989.

Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1987.

Weiner, Richard S. and Lewis J. Pinson. *An Introduction to Object-Oriented Programming and C++*. Reading, MA: Addison-Wesley, 1988.

## Other languages and C

Brown, Douglas L. *From Pascal to C*. Belmont, CA: Wadsworth Publisher, 1985.

Traister, Robert. *AI Programming in Turbo C*. Blue Ridge Summit, PA: Tab Books, Inc, 1989.

# Reference

American National Standard for Information Systems (ANSI). *Programming Language C*. Draft. Document number X3J11/ 88-159. Washington, DC: Computer & Business Equipment Manufacturers Association, 1988.

Barkakati, Naba. *The Waite Group's Essential Guide to Turbo C*. Indianapolis, IA: Howard W. Sams & Co, 1989.

Barkakati, Naba. *The Waite Group's Turbo C Bible*. Indianapolis, IA: Howard W. Sams & Co, 1989.

Bloom, Eric and Jeremy Soybel. *Turbo C Trilogy: A Complete Library for Turbo C Programs*. Blue Ridge Summit, PA: Tab Books, Inc, 1988.

Harbison, Samuel P. and Guy L. Steele. *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Holtz, Frederick. *Turbo C Programmer's Resource Book*. Blue Ridge Summit, PA: Tab Books, Inc., 1987.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, 2nd edition, Englewood Cliffs, NJ: Prentice-Hall, 1988.

O'Brien, Stephen. *Turbo C: The Complete Reference*. Berkeley, CA: Osborne/McGraw-Hill, 1988.

Purdum, Jack and Tim Leslie. *C Standard Library*. Carmel, IN: Que Corporation, 1987.

Rought, Edward R. and Thomas D. Hoops. *Turbo C Developer's Library*. Indianapolis, IA: Howard W. Sams & Co, 1988.

Schildt, Herbert. *Turbo C: The Pocket Reference*. Berkeley, CA: Osborne/McGraw-Hill, 1988.

Schildt, Herbert. *Turbo C/Turbo C++: The Complete Reference*. Berkeley, CA: Osborne/McGraw-Hill, 1990.

# I N D E X

GETTING STARTED

# TURBO C++®

BORLAND

1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300 ■ PART # 14MN-CPP02-10 ■ BOR 1508
UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK, TWYFORD, BERKSHIRE RG10 9NN, ENGLAND
43 AVENUE DE L'EUROPE—BP 6, 78141 VELIZY VILLACOUBLAY CEDEX FRANCE