2.0

# BORLAND® C++

GETTING STARTED


BORLAND

# *Borland® C++*
# Version 2.0

---

## Getting Started

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

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

R1

# C O N T E N T S

# T A B L E S

# F I G U R E S

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

Borland C++ is for professional C++ and C developers who want a powerful, fast, and efficient compiler with which to create just about any application, including Microsoft Windows applications. Also with Borland C++, you get both AT&T's C++ version 2.0 and ANSI C.

*Borland 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 Borland C++

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

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

- C++: Borland 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. We've also included support for C++ version 1.2 streams.
- ANSI C: Borland C++ provides you with an up-to-date implementation of the latest ANSI C standard.

***New!***

- Microsoft Windows targeting: You can use Borland C++ to write applications for Windows. Many features have been added to support this capability, including the Resource Compiler and the Whitewater Resource Toolkit. We've included a few sample C and C++ Windows applications to help get you going.

***New!***

- Precompiled headers, which speed up program compilation time.

***New!***

- Real and protected-mode versions of each compiler. You get four compilers with this product: a real and protected-mode

version of the Programmer's Platform, and a real and protected-mode version of the command-line compiler. Each compiler contains both C and C++ capabilities. Running the compiler in protected mode gives you greater capacity with no swapping.

- A container class library giving you bags, sets, arrays, and so on.
- The Programmer's Platform, Borland's next generation of user interface. The Programmer's Platform, also known as the IDE, provides access to the full range of programs and tools on your computer. Running either in protected or real mode, it includes:
  - a multi-file editor
  - multiple overlapping windows
  - mouse support
  - an integrated debugger
  - ***New!*** a built-in assembler
  - ***New!*** an undo and redo feature with an extensive buffer
  - 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.
- Online hypertext help, with copy-and-paste program examples for practically every function.
- ***New!*** The help now includes the Windows API.
- Many indispensable library functions, including heap checking functions and a complete set of complex and BCD math functions.

Other features include:

- ***New!*** Fast huge arithmetic.
- Far objects and huge arrays.
- 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

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

Borland 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 Borland C++, the 80x87 chip can significantly enhance your programs' performance.

Borland 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. If you have had your mouse for a while, you may want to contact your mouse's manufacturer for the most recent mouse drivers.
- Genus mouse version 9 or later.
- IMSI mouse version 6.11 or later.
- Logitech Mouse version 3.4 or later.
- Mouse Systems' PC Mouse version 6.22 or later.

## Writing for Windows

If you plan to use Borland C++ to create Windows applications, you may want to buy the documentation for Microsoft's Software Developer's Kit (SDK) for Microsoft Windows. Alternatively, you could purchase Charles Petzold's *Programmming Windows*.

# The Borland C++ implementation

Borland 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, Borland C++ includes certain extensions for mixed-language and mixed-model programming that let you exploit your PC's capabilities. See chapters 1 through 4 in the *Programmer's Guide* for a complete formal description of Borland C++.

# The Borland C++ package

Your Borland C++ package consists of a set of disks and five 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 and C++.*

- *Borland C++ Getting Started* (this manual)
- *Borland C++ User's Guide*
- *Borland C++ Programmer's Guide*
- *Borland C++ Library Reference*
- *Whitewater Resource Toolkit*

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

## Getting Started

This volume introduces you to Borland 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 Borland C++ documentation set. These are the chapters in this manual:

**Chapter 1: Installing Borland C++** tells you how to install Borland C++ on your system; it also tells you how to customize the colors, defaults, and many other aspects of Borland C++.

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

**Chapter 3: For Microsoft C users** gives some guidelines on how to convert your Microsoft C 6.0 programs to Borland C++.

*Chapters 4 and 5 work together: The first provides the theory, the other provides the practice.*

**Chapter 4: A C++ primer** is an introduction to the concepts of object-oriented programming using C++.

**Chapter 5: Hands-on C++** provides practical examples based on the concepts introduced in Chapter 4.

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

## The *User's Guide*

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

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

**Chapter 2: Menus and options reference** provides a complete reference to the menus and options in the Programmer's Platform.

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

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

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

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

**Chapter 7: Utilities** describes a few of the many utility programs that come with Borland C++.

## The *Programmer's Guide*

The *Programmer's Guide* provides useful material for the experienced C user: a complete language reference for C and C++, writing Windows applications, 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.

**Chapters 1 through 4: Lexical grammar, Phrase-structure grammar, C++, and The preprocessor**, describe the Borland C++ language.

**Chapter 5: C++ streams** tells you how to use the C++ version 2.0 stream library. The C++ version 1.2 stream library is documented online.

**Chapter 6: Memory management** covers memory models, mixed-model programming, and overlays.

**Chapter 7: Math** covers floating-point and BCD math.

**Chapter 8: Video functions** is devoted to handling text and graphics in Borland C++.

**Chapter 9: Interfacing with assembly language** tells how to write assembly language programs so they work well when called from Borland C++ programs. It includes information on the built-in assembler in the IDE.

**Chapter 10: 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, and how Borland has chosen to implement them.

**Appendix B: 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."

**Appendix C: The container class library** documents the container class library included with Borland C++.

## The *Library Reference*

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

**Chapter 1: The main function** describes the **main** function.

**Chapter 2: The run-time library** is an alphabetically arranged reference to all Borland C++ library functions.

**Chapter 3: Global variables** defines and discusses Borland C++'s global variables.

## The *Whitewater Resource Toolkit*

The *Whitewater Resource Toolkit User's Guide* tells how to create resources for your Windows applications.

**Chapter 1, "Getting started,"** tells you how to begin and end a session in the Resource Toolkit.

**Chapter 2, "About resources and files,"** provides an overview of resources, and discusses the files you can work with in the Resource Toolkit.

**Chapter 3, "The Resource Manager,"** tells you how to use the Resource Toolkit's main window, the Resource Manager.

**Chapters 4 through 9** each tell you how to use a specific Resource Toolkit editor.

**Chapter 10, "Common keys and menus,"** tells you how to navigate and edit tables in the Accelerator, String, and Menu editors. It also tells you how to use the menus whose options are common among the editors: the File, Edit, and Header menus.

**Appendix A, "Troubleshooting and error messages,"** contains questions and answers to help you solve problems you might have while working with the Resource Toolkit.

# Typefaces and icons used in these books

All typefaces and icons used in this manual were produced by Borland's Sprint: The Professional Word Processor, on a PostScript laser printer.

*Monospace type* This typeface represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as `BC` to start up Borland 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*** Borland C++ function names (such as **printf**), class, and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used in text for Borland 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. For example, "Press *Esc* to exit a menu."

This icon indicates keyboard actions.

This icon indicates mouse actions.

This icon indicates language items that are specific to C++. It is used primarily in the *Programmer's Guide*.

This icon indicates material that relates to writing a Windows program.

# How to contact Borland

*Before contacting Borland, read the README and HELPME!.DOC files; many common problems are resolved in them.*

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—Borland C++
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001, 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 a.m. and 5 p.m. Pacific time at (408) 438-5300. 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 Borland 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. (You can find this out by typing `VER` at the 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 Borland C++

*Your Borland C++ package includes four different versions of Borland C++: the IDE in real and protected mode and the real and protected mode versions that can be run from the DOS command line. If you don't already know how to use DOS commands, refer to your DOS reference manual before setting up Borland C++ on your system.*

Borland C++ comes with an automatic installation program called INSTALL. Because we used file-compression techniques, you must use this program; you can't just copy the Borland C++ files onto your hard disk. Instead, INSTALL automatically copies and uncompresses the Borland C++ files. For reference, the README file on the installation disk includes a list of the distribution files.

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 your distribution disks when you receive them, then store the original disks away in a safe place.

None of Borland's products use copy protection schemes. If you are not familiar with Borland's No-Nonsense License Statement, read the agreement included with your Borland 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 Borland C++.

This chapter contains the following information:

- installing Borland C++ on your system
- accessing the README file
- accessing the HELPME! file
- a pointer to more information on Borland's Turbo Calc program
- how to customize Borland C++ (set or change defaults, colors, and so on)

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

## Using INSTALL

*We recommend that you read the README file before installing.*

Among other things, INSTALL detects what hardware you are using and configures Borland 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 Borland C++:

1. Insert the installation disk (disk 1) into drive A. Type the following command, then press *Enter*.

   ```
   A:INSTALL
   ```

2. Press *Enter* at the installation screen.
3. Follow the prompts.
4. At the end of installation, you may want to add this line to your CONFIG.SYS file:

   ```
   FILES = 20
   ```

   and this line to your AUTOEXEC.BAT file:

   ```
   PATH = C:\BORLANDC\BIN
   ```

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

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

Once you have installed Borland C++, and if you're anxious to get up and running, change to the Borland C++ directory and type `BC` (or `BCX`, depending on whether you want to run in real or protected mode) and press *Enter*. Otherwise, continue reading this chapter and the next for important start-up information.

After you have tried out the IDE, you may want to permanently customize some of the options. The BCINST program makes this easy to do; see page 14 for more information.

## 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 Borland C++. The IDE works best if you type `MODE BW80` at the DOS command line before running Borland C++.

Although you could create a batch file to take care of this for you, you can also easily install Borland C++ for a black-and-white screen with the Borland C++ customization program, BCINST. 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 Borland C++, insert your Borland C++ disk into drive A. If you have installed Borland 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 the file, use the ↑ and ↓ keys to scroll through the file.
4. Press *Esc* to exit.

*See Chapter 1, "The Programmer's Platform" in the User's Guide, for more details on using the Programmer's Platform.*

Once you've installed Borland C++, you can open README into an edit window, following these steps:

1. Start Borland C++ by typing `BC` (or `BCX`) on the command line. Press *Enter*.
2. Press *F10*. Choose **F**ile | **O**pen. Type in `README` and press *Enter*. Borland C++ opens the README file in an edit window.
3. When you're done with the README file, choose **F**ile | **Q**uit (or continue playing with the Platform).

# 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 Borland 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.

# Customizing the IDE

Through BCINST, you can change various default settings in the IDE (both the real and protected-mode versions: BC.EXE and BCX.OVY), such as the editing modes, menu colors, and default directories. BCINST also lets you specify and directly modify other .EXE and .PRJ files. If you don't specify a file, BCINST assumes BC.EXE. If BCINST doesn't find the file you specified, it reports an error.

With BCINST, you can do any of the following:

*Borland C++ comes ready to run: You don't need to run BCINST if you don't want to.*

- modify compiler options in a .PRJ file
- set up paths to the directories where your include, library, and output files are located
- choose default settings for the integrated debugger
- set defaults for the compiler and linker
- customize the editor command keys
- set up the editor defaults
- set up the default video display mode
- change screen colors
- bind a key macro to the gray asterisk key (*) on your keyboard in verbatim mode

*For detailed information on the menus and options in the IDE, see Chapter 2, "Menus and options reference," in the User's Guide.*

BCINST's menus are quite similar to the menus in the IDE. Any option that you install with BCINST that *also* appears as a menu option in BC.EXE will be overridden whenever you load a configuration file that contains a different setting for that option, or when you change the setting via the menu system of the IDE. So changes made to BC.EXE are only realized when no configuration files are loaded. For this reason, BCINST lets you directly modify .PRJ files and the TCCONFIG.TC file.

## Running BCINST

The syntax for BCINST is

BCINST [*option*] [*exepath* [*exename*] | [*configpath*] TCCONFIG.TC | [*prjpath*]*prjname.PRJ*]

If you don't give a path and/or file name, BCINST looks for BC.EXE in the current directory. *option* lets you specify whether you want to run BCINST in color (type `/c`) or in black and white (type `/b`). Normally, BCINST comes up in color if it detects a color adapter in a color mode. You can override this default if, for instance, you are using a composite monitor with a color adapter, by using the **/b** option.

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

### Using an EGA card with a CGA monitor

If you are running Borland C++ on a system with an EGA display card and a CGA monitor, you *must* use BCINST to set Borland C++ or it will not run properly. See page 18 for step-by-step instructions on how to do this.

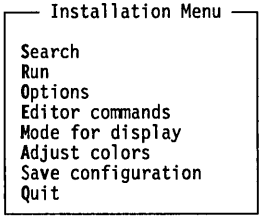
## The BCINST Installation menu

Although BCINST allows you to customize a wide range of Borland C++'s components, BCINST is "smart." That is, it will only show you those menus and selections that apply to what you are customizing. For example, when you're installing a configuration file (.PRJ or .TC), only the menu items representing values in the configuration files are displayed. Therefore, when you're using BCINST to modify the TCCONFIG.TC file, compiler options

are not available; when you're installing a .PRJ file, color customization is not available.

Also, since the BCINST installation menu and option choices correspond with those portions of the IDE that you are modifying, you can refer to Chapter 2, "Menus and options reference," in the *User's Guide* for detailed information on what each item might mean.

The first menu to appear on the screen is the BCINST Installation menu.




- The **S**earch option gives you access to the search defaults.
- The **R**un option allows you to set default command-line arguments that will be passed to your running programs, exactly as if you had typed them on the DOS command line (redirection is not supported). It is only necessary to give the arguments here; you don't need to give the program name.
- The **O**ptions command gives you access to default settings for a great many features, including memory model, degree of optimization, display of error messages, linker and environment settings, and path names to the directories holding header and library files.
- The **E**ditor Commands option lets you customize the interactive editor's keystroke commands. You can restore the default **E**ditor Commands by choosing the *E* option at the BCINST main menu, then press *R* (for **R**estore factory defaults) and *Esc*.
- With **M**ode for Display, you can specify the video display mode that Borland C++ will operate in, and whether yours is a "snowy" video adapter.
- You can customize the colors of almost every part of the IDE through the **A**djust Colors menu.
- The Sa**v**e Configuration option allows you a choice between saving and not saving changes to the BC.EXE file. You can always run BCINST again if you want to change your changes.
- The **Q**uit option asks if you want to quit without saving the changes you have made to the integrated development environment.

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

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

## Some specifics

While for the most part BCINST's menu items are self-explanatory, there are some items that you may need a little more information on.

### Segment names

With the items in the **O**ptions | **C**ompiler | **N**ames menu, you can set the default segment, group, and class names for **C**ode, **D**ata, and **B**SS sections. When you choose one of these items, the asterisk (*) on the next menu that appears tells the compiler to use the default names. *Important!* Don't change this option unless you are an expert and have read Chapter 9 ("Interfacing with assembly language") in the *Programmer's Guide*.

### The Debugger menu

The items in the De**b**ugger menu let you set certain default settings for the Borland C++ integrated debugger.

When you compile your program with **S**ource debugging set **O**n, you can debug it using either the integrated debugger or with a standalone debugger, such as Borland's Turbo Debugger. When it is set to **S**tandalone, only the standalone debugger can be used. When it is set to **N**one, no debugging information is placed in the .EXE file.

**D**isplay Swapping enables you to set the default level to **N**one, **S**mart, or **A**lways. When you run your program with the default setting **S**mart, the Debugger looks at the code being executed to see whether the code will affect the screen (that is, output to the screen). If the code outputs to the screen (or if it calls a function), the screen is swapped from the Editor screen to the Execution screen long enough for output to take place, then is swapped back. Otherwise, no swapping occurs. The **A**lways setting causes the screen to be swapped every time a statement executes. The **N**one setting causes the debugger not to swap the screen at all.

**P**rogram Heap Size specifies the new program heap size in kilobytes, from 4 through 640.

You have several choices for **I**nspector Options: **S**how Inherited (*On* or *Off*), Show **M**ethods (*On* or *Off*), and/or Show **I**ntegers As. Show **I**ntegers As gives you the choice between **D**ecimal, **H**ex, or **B**oth.

### Editor commands

Many of the editor's commands and keystrokes can be customized. Most are self-explanatory. You might need to know that secondary keystrokes take precedence over primary keystrokes, and that there are certain rules governing the keystroke sequences that you can define. Some of the rules apply to any keystroke definition, while others come into effect only in certain keystroke modes.

1. You can enter a maximum of six keystrokes for any given editor command. Certain key combinations are equivalent to two keystrokes: These include *Alt (any valid key)*; the cursor-movement keys (↑, ↓, *PgDn*, *Del*, and so on); and all function keys and their combinations (*F4*, *Shift-F7*, *Alt-F8*, and so on).
2. The first keystroke must be a character that is neither alphanumeric nor punctuation: that is, it must be a control key or a special key.
3. To enter the *Esc* key as a command keystroke, type *Ctrl-[*.
4. To enter the *Backspace* key as a command keystroke, type *Ctrl-H*.
5. To enter the *Enter* key as a command keystroke, type *Ctrl-M*.
6. The predefined Help function keys (*F1* and *Alt-F1*) can't be reassigned as editor command keys. However, any other function key can. If you enter a hot key as part of an editor command key sequence, BCINST issues a warning that you are overriding a hot key in the editor and verifies that you want to override that key.
7. You can assign keys such as the grey asterisk, grey minus, and grey plus by using the Borland C++ editor's verbatim mode.

*Chapter 1, "The Programmer's Platform," in the User's Guide contains a complete list of the IDE's predefined hot keys.*

## Setting your video mode

Normally, Borland C++ correctly detects your system's video mode. You should only change the **M**ode for Display menu if one of the following holds true:

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

If you choose **D**efault, Borland C++ always operates in the mode that is active when you load it.

If you choose **C**olor, Borland C++ uses 80-column color mode if a color adapter is detected, no matter what mode is active when you load the IDE, and switches back to the previously active mode when you exit.

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

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

If you choose **M**onochrome, Borland C++ uses monochrome mode if a monochrome adapter is detected, no matter what mode is active.

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

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

When you press any key, you can choose

- **Y**es, the screen was "snowy"
- **N**o, always turn off snow checking
- **M**aybe, always check the hardware

Look at the Quick-Ref line for more about **M**aybe. Press *Esc* to return to the main installation menu.

C H A P T E R

2

# Navigating the Borland C++ manuals

This chapter accomplishes two things:

- It tells you briefly about Borland 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 Borland C++.

If you read the instructions on how to install Borland C++ on page 12, you also learned how to start Borland 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

Borland 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.

### Windows

With Borland C++, you can write applications that will run under Microsoft's Windows. While the effect of this capability ripples throughout the entire product, certain chapters are devoted to the subject. Chapter 3, "For Microsoft C users" will help you become acquainted with the differences between programming in Microsoft C and programming using Borland C++; we think

you'll be pleased with some of the nice touches in Borland C++ that make your programming task easier. Chapter 3, "Building a Windows application," in the *User's Guide* tells you how to use Borland C++ to write a Windows application.

## C++

With Borland C++, you get all the capabilities of ANSI C, *plus* all the capabilities of C++. You can either program solely in ANSI C, or you can make the transition from C to C++ as slowly or as rapidly as you like.

In order to help you get started, we've included a ready-made set of C++ class libraries. These libraries use classes to perform a variety of functions for you. The streams library is discussed in Chapter 5, "C++ streams," in the *Programmer's Guide*; the container class library is documented in Appendix C, "The container class library," also in the same manual. There are a number of excellent third-party books available on C++ programming, many specific to Borland's C and C++ products. The bibliography in this book (*Getting Started*) lists some of them.

## Real and protected modes

The IDE and command-line compilers come in two versions each: a real-mode version and a protected-mode version. In each of two chapters in the *User's Guide* (chapters 2 and 6, "Menus and options reference" and "The command-line compiler") you'll find a section on how to start up Borland C++ in your preferred mode, and a discussion on which mode might be most appropriate for your needs.

## Built-in assembly language programming

The IDE incorporates a built-in assembly language compiler, called BASM for short. A subset of Turbo Assembler, BASM can do everything Turbo Assembler can do, with a few exceptions. Chapter 9, "Interfacing with assembly language," in the *Programmer's Guide* tells you two things: how to interface your C and C++ programs with assembly language code, and how to use both BASM and Turbo Assembler to do so.

### VROOMM (overlays)

*Chapter 6, "Memory management," in the Programmer's Guide covers overlays in depth.*

Borland 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 Programmer's Platform (IDE)

Borland C++ has Borland's new integrated development environment, called the Programmer's Platform because it allows you to pull in your favorite tools for use within the environment. Chapter 1, "The Programmer's Platform," in the *User's Guide* covers the general appearance and functioning of the IDE. Chapter 2, "Menus and options reference," in that same book provides a reference to every menu and option.

## 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 Borland C++ as a product; the *Programmer's Guide* and the *Library Reference* provide material on programming issues in C and C++.

Chapter 1 of this manual (*Getting Started*) tells you how to install Borland C++ and how to customize Borland C++'s defaults (including its colors). Chapters 4 and 5 provide complementary tutorials (theory and practice) on programming in C++.

The chapters of the *User's Guide* are for use as reference chapters to using Borland C++'s IDE, editor, project manager, command-line compiler, precompiled headers, and online utilities. As well, it provides some information on programming in Windows.

## Programmers learning C or C++

If you are learning C++ but are already familiar with C, you may want to check out chapters 4 and 5. These two chapters work together, one providing the theory, the other providing the practice of writing C++ programs.

If you don't know C, there are many good products on the market that can get you going in that language. The bibliography provides a list of useful books on programming in C, C++, and Borland C++ especially.

In either case, you can use chapters 3 through 7 in the *User's Guide* for reference on specific technical aspects of Borland C++.

Your next step is to start programming in C and C++. You'll find Chapter 2, "The run-time library" in the *Library Reference* to be a valuable reference on how to use each function. Chapter 1, "The main function," provides information on aspects of the **main** function that is seldom found elsewhere. Or, you might prefer to use the online help; it contains much of the same information as the *Library Reference*, and includes programming examples that you can copy into your own programs. Once you have grown comfortable with programming, you may want to move into the more advanced issues covered in the *Programmer's Guide*.

## Experienced C and C++ programmers

If you are an experienced C or C++ programmer and you've already installed Borland C++, you'll probably want to jump immediately to the *Programmer's Guide* and to the *Library Reference*.

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 (Appendix B, "Run-time library cross-reference"). So, for example, if you want to know which 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

# For Microsoft C users

If you're an experienced C or C++ programmer, but the Borland C++ programming environment is new to you, then you should read this short chapter before you do anything else. We appreciate that you want to be up and running fast with a new piece of software, and we know that you want to spend as little time as possible reading the manual. However, the time that you spend reading this chapter will probably save you a lot of time later. Please read on.

## Environment and tools

The Borland C++ IDE (integrated development environment) is roughly the equivalent of the Programmer's Workbench, although naturally we think you'll find the IDE much easier to use. Chapter 2 in the *User's Guide* provides a complete reference to the IDE. If you're interested in building Windows applications, see Chapter 3 in the *User's Guide*.

*You can find out more about configuration and project files in chapters 1 and 4 in the User's Guide.*

The IDE loads its settings from two files: TCCONFIG.TC, the default configuration file, and a project file (.PRJ). TCCONFIG.TC contains general environmental information. The current project file contains information more specific to the application you're building.

A project is the IDE's equivalent of a makefile. It includes the list of files to be built, as well as settings for the IDE options that

control the compilation and linkage of that program. If you don't specify a project file when you start the IDE, a nameless project is opened and set with default compiler and linker options, but no file name list.

Unlike Microsoft C, however, Borland C++ does not automatically create and run a makefile based on settings and file names that you give it in the project. If you want to use the IDE to set up a project, but use MAKE to do the actual build, then you can use the PRJ2MAK utility to convert a project file to a makefile.

The following sections describe the significant differences between Borland C++'s MAKE, Project Manager, linker (TLINK), and command-line compiler (BCC) and Microsoft C's NMAKE, LINK, and CL.

## The IDE and Windows

If you want to have both Windows and the IDE readily available, load TKERNEL and then run Windows in standard mode. *Alt-Esc* switches between the two environments.

To ensure that a project is placed in the correct directory, invoke BCX with a .PIF file which has the Start-up Directory field set to the desired directory for the project. If you don't use a .PIF file, new projects will be placed in the default Borland C++ directory, \BORLANDC\BIN.

## Paths for .h and .LIB files

Microsoft C works with two environment variables, LIB and INCLUDE. The Microsoft linker uses the LIB variable to discover the location of the run-time libraries; similarly, INCLUDE is used to find standard header files. Borland C++ does not use environment variables to store the path for the library or include files. Instead, you can easily set these paths in the IDE using the environment options. If you are working with the command-line compiler, the linker, or the Resource Compiler, you can use command-line options or configuration files.

When you install Borland C++, you are asked to set paths for include files and library files. Those paths are then the default paths in the IDE. The include and library files paths are also written to the default command-line compiler configuration file TURBOC.CFG. The library path is written to the default stand-alone linker configuration file TLINK.CFG.

- In the IDE, reset default search paths for libraries and header files with the **O**ptions | **D**irectories command. The settings in the Directories dialog box become a part of the current project. If you did not start the IDE with a project, or open a new project, the IDE will use a default project.
- For the command-line compiler, you can reset the search path for include and library files with the **–I** and **–L** options, respectively. These options can also be reset in the configuration file for the command-line compiler, TURBOC.CFG.
- For the linker, TLINK or TLINKX, you can use the **–L** option to change search paths for libraries and initialization code (like C0s.OBJ, the startup code for the small memory model). For instance, this option

  ```
  /LC:\BORLANDC\LIB;C:\WINAPPS\LIB
  ```

  tells the linker to look in the two paths named for library and initialization files.

  You can also create a TLINK.CFG file. TLINK.CFG is a regular text file that contains a list of valid TLINK options.
- For the Resource Compiler, the **–x** option tells it to ignore the INCLUDE variable. In addition, you can specify an additional search path with the **–i** option (**-i** all by itself does not imply **–x**).

*Borland C++ licenses the Resource Compiler from Microsoft.*

When the Resource Compiler is invoked from the command line, it looks for windows.h on the path specified by the INCLUDE environment variable, if there is one. If that INCLUDE variable is set to some other path than the location of the windows.h supplied by Borland C++, your module might not be compiled correctly. (This does not occur in the IDE, because the IDE passes the correct information to the Resource Compiler.)

For instance, if you have been using Microsoft C, then you probably have an INCLUDE environment variable set to the path of the Microsoft C header files. If you have also been using the Microsoft Windows SDK, then the version of windows.h included with the SDK is probably also in the INCLUDE directory.

When you're building a Borland C++ application, the Resource Compiler should include the windows.h shipped with Borland C++. If you have a defined INCLUDE environment variable, then you should tell the Resource Compiler to ignore it with the **–x** option.

## MAKE

The MAKE included with Borland C++ is based on the UNIX version of MAKE and is similar to the new Microsoft utility NMAKE. However, the MAKE utility of Microsoft C 5.1 has important differences from Borland C++ MAKE.

The primary difference between the Borland C++ MAKE and the old Microsoft MAKE is that Microsoft MAKE updates all of the targets sequentially. Borland C++ MAKE and NMAKE only update the targets specified on the command line and any targets that occur in their dependency lists. If no targets are supplied on the command line, the first target listed in the makefile is updated.

You can easily convert a Microsoft C 5.1 makefile to the Borland C++ MAKE or Microsoft NMAKE format. If you have a single target that depends on all the others you can move that dependency to be the first dependency.

Another way to convert old makefiles is to create a new pseudo-target as the first target, and have that first target depend on all the other targets in the makefile.

For example, if you have the following makefile:

```
file1.exe: file.obj
   $(LINK) ...

file2.exe: file.obj
   $(LINK) ...

mylib.lib
   $(LIB) ...

file.obj: file.c file.h
   $(CC)
```

Just add as the first target:

```
ALL: file1.exe file2.exe mylib.lib
```

Microsoft C MAKE (version 5.1) allows multiple targets with the same name in a single makefile. You can rewrite these rules into a single rule. Move the dependent files to the dependency list of the first rule, then move the commands to the end of the command list of the first rule.

For instance, Microsoft C MAKE would allow this makefile fragment:

```
whello.exe: whello.obj whello.def
   tlink /Tw /v /n /c C:\BORLANDC\LIB\c0ws whello,\
         whello,\
         ,\
         C:\BORLANDC\LIB\cwins C:\BORLANDC\LIB\cs
C:\BORLANDC\LIB\import,\
         whello

whello.exe: whello.res
   RC whello.res
```

However, the rules shown in this example would not work with Borland C++ MAKE or Microsoft NMAKE. They can be rewritten, like this:

```
whello.exe: whello.obj whello.def whello.res
   tlink /Tw /v /n /c C:\BORLANDC\LIB\c0ws whello,\
         whello,\
         ,\
         C:\BORLANDC\LIB\cwins C:\BORLANDC\LIB\cs
C:\BORLANDC\LIB\import,\
         whello
   rc whello.res
```

Here's a complete list of MAKE's command-line options. Note that for Borland C++, case (upper or lower) *is* significant; the option **–d** is not a valid substitution for **–D**. Case is not significant for Microsoft NMAKE options.

Table 3.1: MAKE and NMAKE options compared

| Microsoft C 6.0 NMAKE option | Borland C++ MAKE option | What it does |
|---|---|---|
| **/?** | **–?** or **–h** | Displays summary of NMAKE syntax. |
| N/A | **–a** | Causes an automatic dependency check on .OBJ files. |
| **/A** | **–B** | Builds all targets regardless of file dates. |
| **/C** | N/A | Does not print copyright messages or nonfatal messages. |
| *Ident*="1" | **–D*Ident*** | Defines the named identifier to the string consisting of the single character 1 (one). |
| ***Ident=String*** | **–D*Ident=String*** | Defines the named identifier *Ident* to *String*. The string cannot contain any spaces or tabs. |
| **/D** | N/A | Prints date/time stamp of accessed file when file is checked. |

Table 3.1: MAKE and NMAKE options compared (continued)

| Microsoft C 6.0 NMAKE option | Borland C++ MAKE option | What it does |
|---|---|---|
| **/E** | | Overrides defined macros with environment variables. (Note that the Borland C++ utilities do **not** use the environment variable INCLUDE and LIB; Borland C++ overrides environment variables with defined macros.) |
| **/F *filename*** | **–f*filename*** | Uses *filename* as the MAKE file. If *filename* does not exist and no extension is given, tries FILENAME.MAK. |
| **/HELP** | (See **–h**) | In Microsoft's NMAKE, this option calls QuickHelp, if available, or displays help onscreen. |
| **/I** | **–i** | Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. |
| N/A | **–I*directory*** | Searches for include files in the indicated directory (as well as in the current directory). |
| N/A | **–K** | Keeps (does not erase) temporary files created by MAKE. All temporary files have the form MAKE*nnnn*.$$$, where *nnnn* ranges from 0000 to 9999. See Chapter 7in the *User's Guide* for more on temporary files. |
| **/N** | **–n** | Prints the commands but does not actually perform them. This is useful for debugging a makefile. |
| **/NOLOGO** | N/A | Does not print NMAKE sign-on banner. |
| **/P** | N/A | Prints macro definitions and target descriptions. |
| **/Q** | N/A | Returns exit code from target (zero if target is current, nonzero if target is out-of-date); useful for batch file invocation of NMAKE. |
| **/R** | N/A | Ignores TOOLS.INI. |
| N/A | **–S** | Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules. |
| **/S** | **–s** | Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed. |
| N/A | **–U*identifier*** | Undefines any previous definitions of the named identifier. |
| N/A | **–W** | Writes the current specified non-string options (like **–s** and **–a**) to MAKE.EXE. (This makes them default.) |
| **/T** | N/A | Changes modification date for out-of-date target files to the current date. (For Borland C++, use the TOUCH utility.) |
| **/X** *FileName* | N/A | Sends error output to *FileName* (file or device). If you enter a dash instead, sends output to standard output. |
| **/Z** | N/A | Internal option for Programmer's Workbench. |

The following table compares NMAKE predefined macros, pseudotargets, and directives with those of Borland C++ MAKE.

Table 3.2: MAKE and NMAKE predefined macros and directives

| Microsoft C NMAKE | Borland C++ MAKE | What it does |
|---|---|---|
| N/A | $d(*Macro*) | Defined test macro. Expands to 1 if *Macro* is defined, 0 if not. See !IFDEF directive, later in this table. |
| N/A | $: | Path only macro. |
| $@ | $. | Full file name macro, no path. |
| $* | $& | Base file name macro, no path. |
| N/A | $* | Base file name macro with path. |
| $** | N/A | List of all dependent files. |
| $< | $< | Full file name macro with path |
| $$@ | N/A | The target currently being evaluated. (Used only in dependency lines.) |
| $(CC) | N/A | Command to invoke command-line compiler. NMAKE predefines this macro to CL |
| $(AS) | N/A | Command to invoke assembler. NMAKE predefines to AS. |
| $(MAKE) | N/A | Command to invoke NMAKE. Used to invoke NMAKE recursively. |
| $(MAKEDIR) | N/A | The directory from which NMAKE was invoked. |
| $(MAKEFLAGS) | N/A | NMAKE options currently in effect. |
| !IF *expression* | !if *expression* | Conditional execution. If *expression* is true, lines following *expression* are executed until !else, !elif (Borland C++ only), or !endif is encountered. |
| !ELSE | !else | Conditional execution. If previous !if *expression* is false, lines following !else are executed until !elif (Borland C++ only), or !endif is encountered. |
| N/A | !elif | Nested conditional execution. |
| !ENDIF | !endif | Ends conditional execution of !if block. For Microsoft C, also ends !IFDEF, and !IFNDEF blocks. |
| !IFDEF *Macro* | N/A | Conditional execution. If *Macro* is defined, lines following !IFDEF are executed until !ELSE or !ENDIF is encountered. See $d(*Macro*) predefined macro for Borland C++ equivalent. |
| !IFNDEF *Macro* | N/A | Conditional execution. If *Macro* is undefined, lines following !IFDEF are executed until !ELSE or !ENDIF is encountered. See $d(*Macro*) predefined macro for Borland C++ MAKE equivalent. |
| !UNDEF *Macro* | !undef *Macro* | Causes the definition for a specified macro to be forgotten. |
| !ERROR *Text* | !error *Text* | Causes MAKE to stop and print an error message. |

Table 3.2: MAKE and NMAKE predefined macros and directives (continued)

| Microsoft C NMAKE | Borland C++ MAKE | What it does |
|---|---|---|
| !INCLUDE *filename* | !include *filename* | Specifies the file *filename* to be included in the makefile. If the form <*filename*> is used, Microsoft NMAKE searches for *filename* in directories specified by the INCLUDE environment variable. For Borland C++ MAKE, use the **-I** option for the same effect. |
| !CMDSWITCHES | N/A | Turns on or off NMAKE's /I (ignore return code), /D (print date/time), /N (print commands; don't execute), or /S (silent) options. For Borland C++, use .ignore, .silent, or .nosilent. (There is no equivalent for NMAKE's /I.) |
| .IGNORE: | .ignore | Ignore return value of commands. |
| N/A | .noignore | Turns off .ignore. |
| N/A | .autodepend | Turns on autodependency checking. |
| N/A | .noautodepend | Turns off autodependency checking. |
| .SILENT: | .silent | Don't print commands before executing them. |
| N/A | .nosilent | Tells MAKE to print commands before executing them. |
| N/A | .swap | Tells MAKE to swap itself in and out of memory. |
| N/A | .noswap | Tells MAKE to not swap itself in and out of memory. |
| N/A | .path.ext | Gives MAKE a path to search for files with extension *.ext*. |
| .SUFFIXES: *ExtList* | N/A | If no dependents are listed for a target, NMAKE tries files with an extension listed in *ExtList*. |
| .PRECIOUS: *Targets* | N/A | Tells NMAKE to save named *Targets*, even if program building target is interrupted. |

## Command-line compiler

The following table lists comparable BCC and CL command-line compiler options. Some of the CPP (standalone preprocessor) options are listed. In many multi-pass compilers, a separate pass performs the work of the preprocessor, and the results of the pass can be examined. Since Borland C++ uses an integrated single-pass compiler, we provide the standalone utility CPP to supply the first-pass functionality found in other compilers.

Note that most CL options that take arguments allow for a space between the option and the argument. BCC options that take arguments are usually immediately followed by the argument or list.

Table 3.3: CL and BCC options compared

| Microsoft C CL option | Borland C++ BCC option | What it does |
|---|---|---|
| N/A | ***@filename*** | Gives the command-line compiler a response file name. |
| N/A | ***+filename*** | Tell the command-line compiler to use the alternate configuration file *filename*. |
| N/A | **–AK** | Use only Kernighan and Ritchie keywords. |
| N/A | **–AU** | Use only UNIX keywords. |
| (See **/Zp*n***) | **–a** | Align word. |
| (See **/Zp*n***) | **–a–** | Align byte (default). |
| **/Aw /Gw** | **–WD** | Creates an .OBJ for Windows to be linked as a .DLL with all functions exportable. |
| **/Aw /GW** | **–WDE** | Creates an .OBJ for Windows to be linked as a .DLL with explicit export functions. |
| **/A*x*** | **–m*x*** | Use memory model *x*. For BCC, following t, s, or m with ! tells compiler to assume DS != SS. |
| **/B*n*** | N/A | Use alternate preprocessor C*n*L. |
| N/A | **–B** | Compile and call the assembler to process inline assembly code. |
| N/A | **–b** | Make enums word-sized by default. |
| N/A | **–b–** | Make enums signed or unsigned. |
| **/C** | **–C** | Nested comments on. |
| **/c** | **–c** | Compile to .OBJ but do not link. |
| **/D*id*** | **–D*name*** | Define *name* to the string consisting of the null character. |
| **/D*id=value*** | **–D*name=string*** | Defines *name* to *string*. |
| N/A | **–d** | Merge duplicate strings on. |
| N/A | **–d–** | Merge duplicate strings off (default). |
| N/A | **–E*filename*** | Use *filename* as the assembler to use. |
| **/E** | CPP **–P** | Preprocess source to standard output, include line numbers. |
| **/EP** | CPP **–P–** | Preprocess source to standard output, without line numbers. |
| N/A | **–f–** | Don't do floating point. |
| N/A | **–ff** | Fast floating point (default). |
| N/A | **–ff–** | Strict ANSI floating point. |
| N/A | **–f87** | Use 8087 hardware instructions. |
| N/A | **–f287** | Use 80287 hardware instructions. |
| **/F *hexnum*** | | Sets stack size to *hexnum* bytes (hexnum must be hexadecimal). |
| (By default) | **–Fc** | Generates COMDEFs. |
| N/A | **–Fm** | Enables the **–Fc**, **–Ff**, and **–Fs** options. |
| (By default) | **–Fs** | Make DS = SS for all memory models. |
| **/Fa [*listfile*]** | N/A | Create assembly listing. Name for list file defaults to *Source*.EXT. |
| **/Fb*bound-exe*** | | Creates a bound executable file. |
| **/Fc [*listfile*]** | **–s** | Produces a combined source and assembly code listing. Name for list file defaults to *Source*.COD. |
| **/Fe *exefile*** | **–e*exefile*** | *exefile* names executable file. |
| **/Fl [*listfile*]** | N/A | Creates object code list. Name for list file defaults to *Source*.COD. |
| **/Fm [*mapfile*]** | **–M** | Creates map file. Name defaults to *Source*.MAP, where source is the first source file specified. |
| **/Fo *objfile*** | **–o*objfile*** | *objfile* names object file. |

Table 3.3: CL and BCC options compared (continued)

| Microsoft C CL option | Borland C++ BCC option | What it does |
|---|---|---|
| **/FPa** | N/A | Generate floating-point calls; select alternate math library. |
| **/FPc** | **–f** | Emulate floating point (default for Borland C++); coprocessor used if present at run time). |
| **/FPc87** | N/A | Selects 80*x*87 library (80*x*87 coprocessor must be present at run time). |
| **/FPi** | N/A | Inlines 80*x*87 instructions; selects emulator library (coprocessor used if present at run time). |
| **/FPi87** | **–f87** or **–f287** | Inlines 80*x*87 instructions; chooses coprocessor library (coprocessor must be present at run time). |
| **/Fr [*browsefile*]** | N/A | Generates standard PWB Source Browser database. |
| **/FR [*browsefile*]** | N/A | Generates extended PWB Source Browser database. |
| **/Fs [*listfile*]** | N/A | Produce source list file. Source list file name defaults to *Source*.LST. |
| **/Fx [*xreffile*]** | N/A | *xreffile* specifies a name for the MASM cross-reference file. |
| **G0** | **–1** | Generate 80186 instructions. |
| **G1** | **–1–** | Generate 8088/8086 instructions. |
| **G2** | **–2** | Generate 80286 protected-mode compatible instructions. |
| N/A | **–G** | Optimize for speed. |
| N/A | **–G–** | Optimize for size (default). |
| **/Gc** | **–p** | Use Pascal calling convention. For CL, this is Pascal or FORTRAN, but currently same calling convention. |
| **/Gd** | **–p–** | Standard C calling conventions (default). |
| **/Ge** | **–N** | Check for stack overflow. (Default for CL, but not for BCC). |
| **/Gi** | N/A | Compile incrementally (for use with quick compile option **/qc**). |
| **/Gm** | N/A | Store strings in CONST segment. |
| **/Gr** | N/A | Enables **_fastcall** to call conventions for functions (if possible, passing value in registers). |
| **/Gs** | **–N–** | Turn off checking for stack overflow. (Off by default for BCC.) |
| **/Gt [*number*]** | **–Ff[=*size*]** | Creates far variables automatically; *size* or *number* is threshold. |
| **/Gw** | **–W** | Creates correct prolog/epilog for Windows program (for Borland C++, this creates an application with all functions exportable). |
| **/GW** | **–WE** | Generates prolog/epilog for explicit functions (marked with _export) in Windows program. |
| N/A | **–H** | Causes the compiler to generate and use precompiled headers. |
| N/A | **–H–** | Turns off generation and use of precompiled headers (default). |
| N/A | **–Hu** | Tells the compiler to use but not generate precompiled headers. |
| N/A | **–H=*filename*** | Sets the name of the file for precompiled headers. |
| By default | **–h** | Use fast huge pointer arithmetic. |
| **/H *number*** | **–i*number*** | Restricts length of external names to *number*. |
| **/HELP** | **BCC** | Calls QuickHelp. For help on BCC, simply invoke without options. |
| N/A | **–i*n*** | Make significant identifier length to be *n*. |
| **/I *directory*** | **–I*path*** | Directories for include files. For CL, adds *directory* to the beginning of include file search directory list. See page 26. |
| N/A | **–j*n*** | Errors: Stop after *n* messages. |
| **/J** | **–K** | Changes default for **char**. from signed to unsigned. For Borland C++, **–K–** returns to signed. |
| N/A | **–k** | Standard stack frame on (default). |
| N/A | **–L*path*** | Directories for libraries. |
| **/Lc** and **/Lr** | **/Td** | Tells linker to create a real mode executable. |

Table 3.3: CL and BCC options compared (continued)

| Microsoft C CL option | Borland C++ BCC option | What it does |
|---|---|---|
| **/Li [*number*]** | N/A | Use incremental linker, instead of standard linker. *Number* specifies byte boundary for padding near functions. |
| **/Lp** | N/A | Create protected mode executable (OS/2). |
| **/Lr** | | See **/Lc**. |
| **/link *options*** | **–l*options*** | Pass *options* to linker when invoked. |
| N/A | **–l–*option*** | Suppress option *option* for the linker. |
| N/A | **–M** | Instruct the linker to create a map file. |
| **/MA*option*** | **–T*option*** | Pass to assembler when invoked. |
| **/MD** | N/A | Creates a DLL for OS/2. |
| **/ML** | N/A | Statically links a library to a DLL (OS/2). |
| **/MT** | N/A | Provides support for multithread programs for OS/2. |
| N/A | **–n*path*** | Set the output directory. |
| **/ND*dataseg*** | **–zR*name*** | Sets the data segment name. For BCC, this option changes the name of the uninitialized data segment class to *name*. By default, the uninitialized data segments are assigned to class BSS. |
| **/NM*module*** | N/A | Sets the module name to *module*. |
| **/nologo** | N/A | Don't print sign-on banner. |
| **/NT*segname*** | **–zC*name*** | Sets code segment name. This option changes the name of the code segment to *name*. By default, the code segment is named _TEXT, except for the medium, large and huge models, where the name is *filename_TEXT*. (*filename* here is the source file name.) |
| N/A | **–O** | Optimize jumps. |
| N/A | **–O–** | No optimization (default). |
| **/O [*options*]** | (See comment) | Provides optimization. For Borland C++, see specific options; for instance, –Z, –O, or –G. |
| N/A | **–P** | Perform a C++ compile regardless of source file extension. |
| N/A | **–P*ext*** | Perform a C++ compile and set the default extension to *ext*. |
| N/A | **–P–** | Perform a C++ or C compile depending on source file extension (default). |
| N/A | **–P–*ext*** | Perform a C++ or C compile depending on extension; set default extension to *ext*. |
| N/A | **–p–** | Use C calling convention (default). |
| **/P** | CPP **–o*filename*** | Preprocesses source file and sends output to *filename* (CPP), or to *Source*.I (CL). |
| N/A | **–Qe** | Instructs the compiler to use all available EMS memory (default). |
| N/A | **–Qe–** | Instructs the compiler to not use any EMS memory. |
| N/A | **–Qx** | Instructs the compiler to use all available extended memory. |
| N/A | **–Qx=*nnnn*** | Instructs the compiler to reserve *nnnn* Kb of extended memory for other programs, and to use the rest itself. |
| N/A | **–Qx=*nnnn,yyyy*** | Instructs the compiler to reserve *nnnn* Kb of extended memory for other programs and *yyyy* for itself. |
| N/A | **–Qx=,*yyyy*** | Instructs the compiler to reserve *yyyy* Kb of extended memory for itself. |
| N/A | **–Qx–** | Instructs the compiler to not use any extended memory |
| N/A | **–r** | Use register variables on (default). |
| N/A | **–r–** | Suppresses the use of register variables. |
| N/A | **–rd** | Only allow declared register variables to be kept in registers. |
| **/qc** | N/A | Invokes quick compile. |

Table 3.3: CL and BCC options compared (continued)

| Microsoft C CL option | Borland C++ BCC option | What it does |
|---|---|---|
| **/S*x option*** | N/A | Set options for source listing. Where *x* is l, p, s, or t. |
| N/A | **–T–** | Remove all previous assembler options. |
| **/Ta *asm_srcfile*** | N/A | Specifies that asm_srcfile be treated as an assembler source file. |
| **/Tc *c-srcfile*** | N/A | Specifies that c_srcfile be treated as a c source file. |
| N/A | **–u** | Generate underscores (default). |
| N/A | **–u–** | Disable underscores. |
| **/u** | N/A | Undefines all predefined identifiers. |
| **/U *Ident*** | **–U*Ident*** | Undefine any previous definitions of *Ident*. |
| N/A | **–V** | Smart C++ virtual tables. |
| N/A | **–Vs** | Local C++ virtual tables. |
| N/A | **–V0, –V1** | External and Public C++ virtual tables. |
| N/A | **–Vf** | Far C++ virtual tables. |
| N/A | **–vi, –vi–** | Controls expansion of inline functions. |
| **/V *string*** | N/A | Copies *string* to object file (for version control). |
| N/A | **–w** | Display warnings on. |
| N/A | **–w*xxx*** | Enable *xxx* warning message. |
| N/A | **–w–*xxx*** | Disable *xxx* warning message. |
| **/w** | **–w–** | Display warnings off. |
| N/A | **–WS** | Creates an .OBJ for Windows that uses smart callbacks. |
| **/W *n*** | (See **–w**) | Set warning level 0, 1, 2, 3, or 4. |
| **/WX** | **–g1** | Makes all warnings fatal. No object files are generated if warning occurs. (The –g option takes the form –g*n*, where *n* is the limit to number of warnings.) |
| N/A | **–X** | Disable compiler autodependency output. |
| **/X** | N/A | Ignore INCLUDE environment variable list of include search paths. |
| N/A | **–Y** | Enable overlay code generation. |
| N/A | **–Yo** | Overlay the compiled files. |
| N/A | **–Z** | Enable register usage optimization. |
| N/A | **–zA*name*** | Code class. |
| N/A | **–zB*name*** | BSS class. |
| N/A | **–zD*name*** | BSS segment. |
| N/A | **–zE*name*** | Far segment. |
| N/A | **–zF*name*** | Far class. |
| N/A | **–zG*name*** | BSS group. |
| N/A | **–zH*name*** | Far group. |
| N/A | **–zP*name*** | Code group. |
| N/A | **–zS*name*** | Data group. |
| N/A | **–zT*name*** | Data class. |
| N/A | **–z*X**** | Use default segment, class, or group name for *X*. |
| **/Za** | **–A** | Enforces ANSI compatibility. Use only ANSI keywords. No vendor-specific extension allowed. |
| **/Zc** | N/A | Ignores case for functions declared as **_pascal**. |
| **/Zd** | **/y** | Generates line numbers for symbolic debugger. |
| **/Ze** | **–A–, –AT** | Enable vendor-specific extensions. |
| **/Zg** | N/A | Generates function prototypes; writes to standard output. |
| **/Zi** | **/V** | For Microsoft, generates debugger information for CodeView. For Borland C++, generates information for IDE debugger and Turbo Debugger. |

Table 3.3: CL and BCC options compared (continued)

| Microsoft C CL option | Borland C++ BCC option | What it does |
|---|---|---|
| **/Zl** | N/A | Library search records not written to object file. |
| **/Zp*n*** | (See **-a**, **-a-**) | Packs structure members on the *n* byte boundary. *n* can be 1, 2, or 4. |
| **/Zr** | N/A | Generates checks for null pointers and far pointers that are out of range. |
| **/Zs *sourcefiles*** | N/A | Syntax check only. |

## Compatibility command-line options and libraries

The C0F*x*.OBJ modules are provided for compatibility with source files intended for compilers from other vendors. The C0F*x*.OBJ modules substitute for the C0*x*.OBJ modules; they are to be linked with DOS applications only, not Windows applications or DLLs. These initialization modules are written to alter the memory model such that the stack segment is inside the data segment. The appropriate C0F*x*.OBJ module will be used automatically if you use either the **-Fs** or the **-Fm** command-line compiler option.

The **-Fc** (generate COMDEFs), **-Ff** (create far variables), **-Fs** (assume DS = SS in all models), and **-Fm** (enable all **-F*x*** options) command-line compiler options are provided for compatibility. These options are documented in full in Chapter 6 in the *User's Guide*.

## Linker

The Borland C++ linker, TLINK, is invoked automatically from the command-line compiler unless the **-c** compiler option is used. Options such as memory model and target (Windows or DOS), are passed from the compiler to TLINK; TLINK links the appropriate libraries based on the compile options.

TLINK can be used to build both DOS and Windows programs. See the *User's Guide*, Chapter 7, for material on module definition file statements.

The following table compares TLINK and LINK options. Note that Borland C++ TLINK options are case-sensitive, while Microsoft TLINK options are not.

Table 3.4: LINK and TLINK options compared

| Microsoft C 6.0 Link option | Borland C++ TLINK option | What it does |
|---|---|---|
| N/A | /3 | Enable 32-bit processing. |
| /A:*size* | /A=*nnnn* | Specify segment alignment for NewExe (Windows) images. |
| /BA | N/A | BATCH. Suppresses prompts for library or object files not found. |
| N/A | /C | Treat EXPORTS and IMPORTS section of module definition file as case sensitive. |
| /CO | /v | Include full symbolic debug information. |
| /CP:*bytes* | N/A | Sets the program's maximum memory allocation to *bytes*. |
| N/A | /d | Warn if duplicate symbols in libraries. |
| /DOSSEG | (See comment) | For assembly programs, forces a certain ordering of segments in executable. To enable DOSSEG for an assembly program, include DOSSEG in the source code. |
| /DS | N/A | For assembly programs, tells linker to load data starting at high end of DS instead of low end. |
| /E | N/A | Packs the executable by removing repeated series of bytes. |
| /F | By default | For LINK, tells linker to optimize far calls to procedures in same segment as caller. (Used with MS /PACKCODE option.) For TLINK optimizes far calls automatically. |
| /HE | /? | Provides help on command-line options. |
| /HI | N/A | For real-mode assembly programs, places executable as high in memory as possible. |
| N/A | /i | Initialize all segments. |
| /INC | N/A | Prepares for ILINK. |
| /INF | N/A | Tells LINK to display link information while in process. |
| N/A | /L*paths* | Specify library search paths. |
| /LI | /l | Include source line numbers and associated addresses in map file. |
| /M | /m | Create map file with public global symbols. |
| /NOD[:*filename*] | /n | Don't use default libraries. |
| /NOE | /e | Ignore Extended Dictionary. |
| /NOF | N/A | Turns off far call translation (see LINK /F option). |
| /NOI | /c | Treat case as significant in symbols. |
| /NOL | N/A | Causes LINK to suppress banner (logo). |
| /NON | N/A | Arrange segments in executable in the same order as they are arranged by /DOSSEG. |
| /NOP | /P– | Turn off code packing. |
| N/A | /o | Overlay following modules or libraries. Microsoft LINK uses parentheses around files to be overlaid. (Note that the overlay scheme is different between products.) |
| /O:*number* | N/A | Set interrupt *number* for passing control to overlays (other than the default 63). |
| /PACKC[:*number*] | /P=*n* | Pack code segments. *number* or *n* specifies maximum size of groups formed by /PACKC or /P. |
| /PACKD[:*number*] | N/A | Pack data segments. *number* specifies maximum size of groups formed by /PACKD. |
| /PADC:*padsize* | N/A | Tells LINK to pad code module for ILINK. |
| /PADD:*padsize* | N/A | Tells LINK to pad data segments by *padsize* bytes. |
| /PAU | N/A | Pauses linking. |
| /PM:type | N/A | Sets window type for Presentation Manager. |

Table 3.4: LINK and TLINK options compared (continued)

| Microsoft C 6.0 Link option | Borland C++ TLINK option | What it does |
|---|---|---|
| /Q | N/A | Produces Quick library. |
| N/A | /s | Create detailed map of segments. |
| /SE:*number* | N/A | Sets maximum number of segments allowed. |
| /ST:*number* | N/A | Sets stack size. |
| /T | /t | Produce .COM files. |
| N/A | /Td | Create target DOS executable. |
| N/A | /Tdc | Create target DOS .COM file. |
| N/A | /Tde | Create target DOS .EXE file. |
| N/A | /Tw | Create target Windows executable (.DLL or .EXE). |
| N/A | /Twe | Create target Windows application (.EXE). |
| N/A | /Twd | Create target Windows DLL (.DLL). |
| /W | N/A | Warn fixups. |
| N/A | /x | Don't create map file. |
| N/A | /ye | Use expanded memory for swapping. |
| N/A | /yx | Use extended memory for swapping. |

# Source-level compatibility

The following sections tell you how to make sure that your code is compatible with Borland C++'s compiler and linker.

## __MSC

If a library function exists in both the Microsoft and the Borland C++ libraries but it has a different name or a slightly different signature, Borland C++ will substitute the Microsoft function with its equivalent Borland C++ function. It will only do this if **__MSC** is defined. For compatibility, define **__MSC** before any header file is included in any of the source files.

## Header files

Some nonstandard header files can be included by one of two names, as follows.

| Original name | Alias |
|---|---|
| alloc.h | malloc.h |
| dir.h | direct.h |
| mem.h | memory.h |
| varargs.h | (new to this version) |
| search.h | (new to this version) |

If you are defining data in header files in your program, you should use the **–Fc** command-line compiler option to generate COMDEFs. Otherwise you will get linker errors. Chapter 6 of the *User's Guide* provides a complete reference to the command-line compiler options.

## Memory models

Although the same names are used for the six standard memory models, there are fairly significant differences for the large data models in the standard configuration.

In Microsoft C, all large data models (compact, large, and huge) have a default NEAR data segment to which DS is maintained. Data is allocated in this data segment if the data size falls below a certain threshold, or in a far data segment otherwise. You can set the threshold value with the **–Gt***n* option, where *n* is a byte value. The default threshold is 32,767. If **–Gt** is given but *n* is not specified, the default is 256.

In all other memory models under Microsoft C, both a near and a far heap are maintained.

In Borland C++, the large and compact models (but not huge) have a default NEAR data segment to which DS is maintained. All static data is allocated to this segment by default, limiting the total static data in the program to 64K, but making all external data references near. In the huge model all data is far.

In Microsoft's version of the huge memory model, a default data segment for the entire program is maintained which limits total near data to 64K. No limit is imposed on array sizes since all extern arrays are treated as huge (**_huge**).

In Borland C++'s huge memory model, each module has its own data segment. The data segment is loaded on function entry. All data defined in a module is referenced as near data and all extern data references are far. The huge model is limited to 64K of near data in each module.

## Keywords

Borland C++ supports the same set of keywords as Microsoft C 5.1 with the exception of **fortran**.

Borland C++ supports the same set of keywords as Microsoft C 6.0 with the exception of:

- **_based**, **_self**, and **_segname**, because Borland C++ does not support based pointers
- **_segment**; Borland C++'s keyword **_seg** is the equivalent of **_segment**
- **_fastcall**, an optimization for which there is no direct equivalent
- **_emit**; Borland C++ uses the pseudofunction _ _emit_ _(), because this style allows addresses of variables to be given as arguments, and allows multiple bytes to be output; **_emit**, by contrast, works like an assembly DB, allowing one immediate byte to be output
- **_fortran**; use the **_pascal** calling convention instead

Borland C++ provides **_cs**, **_ds**, **_es**, and **_ss** pointer types. See the section "Mixed model programming: Addressing modifiers" in the *Library Reference*, Chapter 6 for more information.

### Intrinsic functions

Borland C++ does not support intrinsic functions.

## Register conventions

Borland C++ and Microsoft C both require the called routine to preserve DS, SS, SI, DI, and BP. Microsoft C requires that the state of the direction flag be preserved across function calls. Borland C++ currently doesn't make any assumptions about the state of the direction flag.

## Floating-point return values

In Microsoft C, **_cdecl** causes float and double values to be returned in the _ _**fac** (floating point accumulator) global variable. Long doubles are returned on the NDP stack. **_fastcall** causes floating point types to be returned on the NDP stack. **_pascal** causes the calling program to allocate space on the stack and pass address to function. The function stores the return value and returns the address.

In Borland C++, floating point values are returned on the NDP stack.

## Structures returned by value

In a Microsoft C-compiled function declared with **_cdecl**, the function returns a pointer to a static location. This static location is created on a per-function basis. For a function declared with **_pascal**, the calling program allocates space on the stack for the return value. The calling program passes the address for the return value in a hidden argument to the function.

Borland C++ returns 1-byte structures in AL, 2-byte structures in AX and 4-byte structures in AX and DX. For 3-byte structures and structures larger than 4-bytes, the compiler passes a hidden argument (a far pointer) to the function that tells the function where to return the structure.

C H A P T E R

# 4

# A C++ primer

*This chapter covers the basic ideas of C++; Chapter 5, "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 disks 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.)

Borland 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. 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 learning C++ much easier 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 Borland 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 in particular 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 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 Borland 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**.

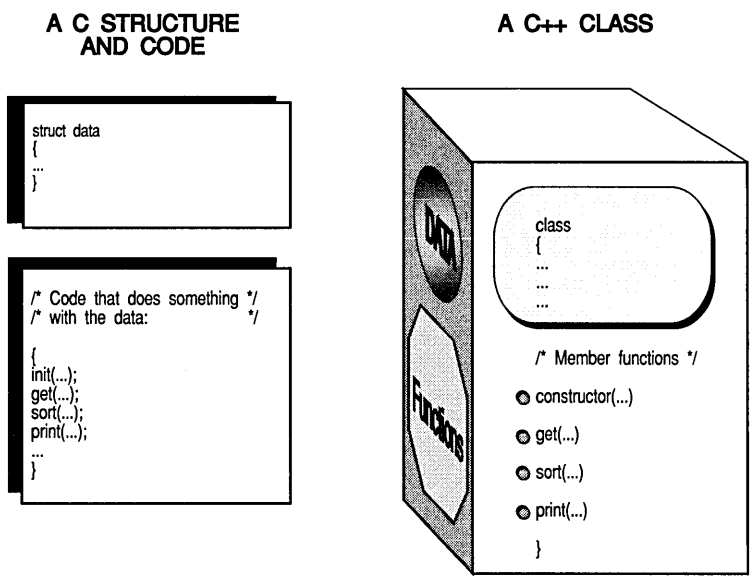
Typically, you restrict member-data access to member functions: you usually make the member data **private** and the member functions **public**.

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 4.1
C versus C++




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.

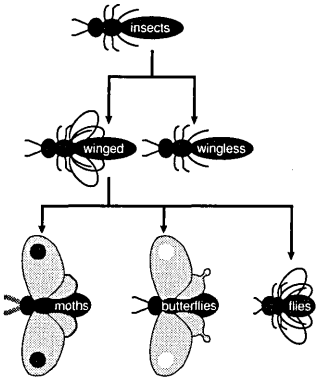
The class concept leads to the idea of *data abstraction*. Our font data structure is no longer tied to any particular physical implementation; rather, it is defined in terms of the operations (member functions) allowed on it. At the same time, the C philosophy that views a program as a collection of *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 4.2. 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 4.2
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 Borland 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 61 and 97.*

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 3, "C++," 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 non-object-oriented languages is *initialization*: Before using a data structure, you must initialize it and allocate memory for it. 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 than 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 older versions of 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 63).

**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 IDE, 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 Borland 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 function, **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 78) 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 106 and also in Chapter 5, "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 4 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 in 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 **union**s 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 4.1
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 |
|---|---|---|
| private | private | not accessible |
| protected | private | private |
| public | private | private |
| private | public | not accessible |
| protected | public | protected |
| public | public | public |

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 3, "C++," 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 4, "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 7, "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 64 are put in the file point.h (on your distribution disks).

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 4 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 3, "C++," 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 disks 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 disks. (Read Chapter 4, "Managing multi-file projects," in the User's Guide if you don't know how to use project files.)*

```
/* PIXEL.CPP--Example from Chapter 4 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 4 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, "..\\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 **friend**s 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 disks 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.

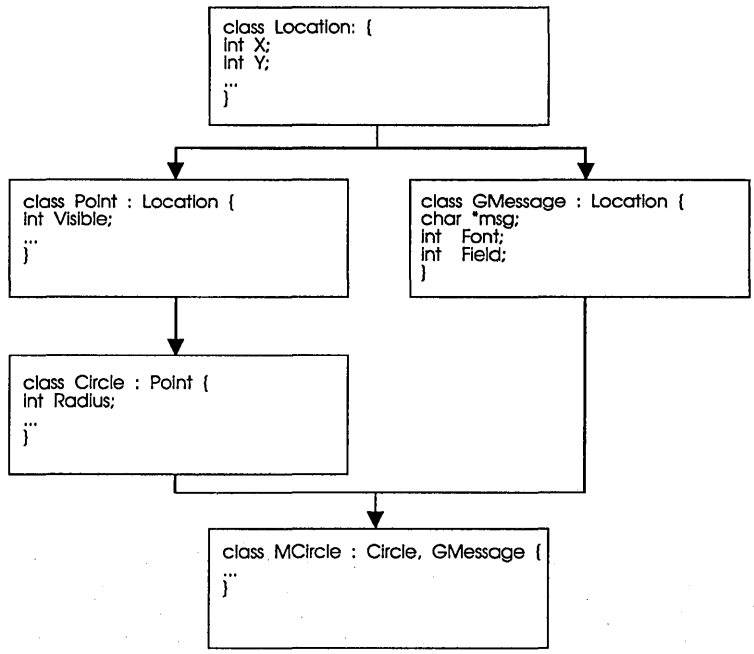
## Multiple inheritance

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 4.3
Multiple inheritance




*This code is available on your disks: MCIRCLE.CPP. You need to run it using MCIRCLE.PRJ.*

```
/* MCIRCLE.CPP--Example for Chapter 4 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);      // magnify if size > 1
    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, "..\\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 3, "C++," 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 3, "C++," in the Programmer's Guide for details on constructor calling sequences.*

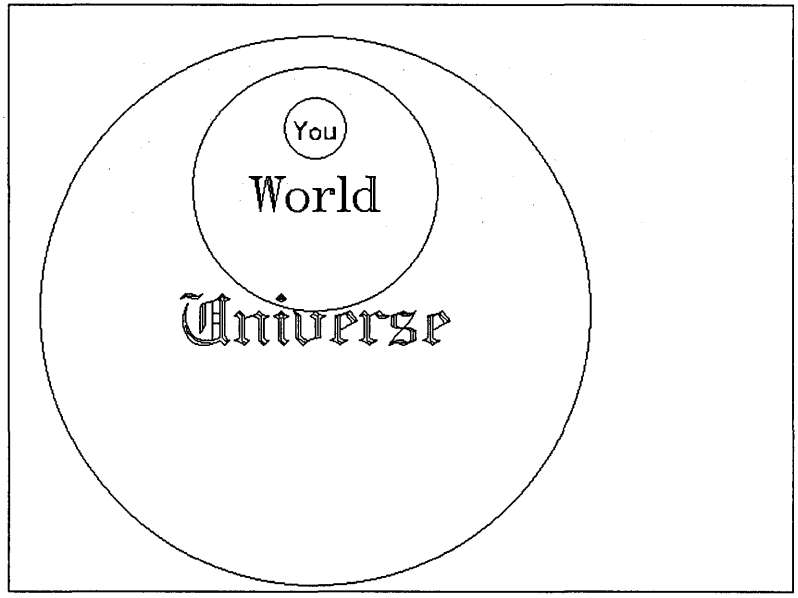
First, the **Circle** constructor is called. This constructor then calls the **Point** constructor, which in turn calls the **Location** constructor. Finally, the **GMessage** constructor is called, which calls the **Location** constructor for its own copy of its base class *X* and *Y*. The arguments 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 4.4 shows the output of MCIRCLE:

Figure 4.4
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 disks) 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 disks), 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 trade-offs: 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 97). 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 what you might be used to in any program.

*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 4

// 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 4, 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

**delete** *pointer;*

*You can find more on destructor syntax in Chapter 3, "C++," in the Programmer's Guide.*

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 4 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, "Lexical grammar," 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 50. (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 I User Screen menu item.*

```
//STRING.CPP--Example from Chapter 4 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 4 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)
{
  String Temp( length + Arg.length );
  strcpy(Temp.char_ptr, char_ptr);
  strcat(Temp.char_ptr, Arg.char_ptr);
  return Temp;
}

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 I 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 5, "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.

Borland 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 Borland C++. However, given a choice, you should convert to the more efficient **iostream** and avoid **stream** when writing

new code. Chapter 5, "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 5, "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>

int main(void)
{
   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 5, "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 4 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:

```
(dest && source.get(ch))
```

*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.*

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 85 (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

# 5

# 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 4, "A C++ primer." You might also want to refer to Chapter 3, "C++," 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 Borland 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 5 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 5 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*

```
// ex3.cpp:   Inline Functions
// from Chapter 5 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 5 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.*

*In an important change from C, declarations can appear anywhere a statement can.*

```
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 5 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 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).

*In other object-oriented languages, member functions are often called **methods**.*

The implementation is usually kept in a separate file:

```
// def.cpp:   Implementation of the Definition class
// from Chapter 5 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 5 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

We can now define a dictionary as a collection of definitions.

*From the command line, build DICTION.OBJ and DEF.OBJ with EX6.CPP. From the Programmer's Platform, use the EX6.PRJ project file.*

```
// diction.h:   The Dictionary class
// from Chapter 5 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;}; // The destructor
   void add_def(char *s, char **def);
   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 5 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 5 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 5 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 5 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 5 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 5 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

#include "list.h"

class Stack : public List
{
   int top;

public:
```

```
    Stack() {top = 0;};
    Stack(int n) : List(n) {top = 0;};
    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 5 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 5 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 5 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.

```
// stack2.cpp:     Implementation of the Stack class

#include <iostream.h>
#include "stack2.h"

int Stack::push(int elem)
{
   if (top < nmax)
   {
      list[top++] = elem;
      return 0;
   }
   else
      return -1;
}

int Stack::pop(int& elem)
{
   if (top > 0)
   {
      elem = list[--top];
      return 0;
   }
   else
      return -1;
}

void Stack::print()
{
   for (int i = top-1; i >= 0; --i)
      cout << list[i] << "\n";
}
```

And then we can try it out:

```
// ex9.cpp:  Using the print() virtual function
// from Chapter 5 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. 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 4, "A C++ primer." Chapter 3, "C++," in the *Programmer's Guide* gives more advanced material on C++. And check the bibliography; it provides a list of books on C++, many specific to Borland C++.

# Bibliography

Many leading book publishers support Borland products with a wide range of excellent books, serving everyone from beginning programmers to advanced users. This bibliography lists primarily books that are specific to Turbo C++; however, much of what is in those books will also apply to Borland C++.

## Beginning to intermediate


Burnap, Steve. *COMPUTE's Turbo C for Beginners*. Radnor, PA: COMPUTE! Publications, 1988.

Chui, Paul and Greg Voss. *Turbo C++ DiskTutor*. Berkeley, CA: Osborne/McGraw-Hill, 1990.

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.

Flamig, Bryan and Keith Weiskamp. *Turbo C++: A Self-Teaching Guide*. New York, NY: John Wiley & Sons, 1990.

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++*. Alameda, CA: Sybex, Inc, 1990.

Ladd, Scott. *Turbo C++ Programming*. Carmel, IN: Que Corproation, 1990.

Ladd, Scott. *Turbo C++ Techniques and Applications*. Redwood City, CA: M & T Books, 1990.

LaFore, Robert. *The Waite Group's C Programming Using Turbo C++*. Indianapolis, IA: Howard W. Sams & Co, 1990.

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.

Smith, Norman. *Illustrated Turbo C++.* Worldware Publishing, 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.

Ezzell, Ben. *Graphics Programming in Turbo C++: An Object-Oriented Approach.* Reading, MA: Addison-Wesley, 1990.

Ezzell, Ben. *Object-Oriented 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.

Lane, Alex. *Turbo C++ By Example*. Redwood City, CA: M & T Books, 1990.

Mosich, Donna, Namir Shammas, and Bryan Flamig. *Advanced Turbo C Programmer's Guide*. New York, NY: John Wiley & Sons, 1988.

Pappas, Chris H. and William H. Murray III. *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.

Weiskamp, Keith. *Object-Oriented Programming with Turbo C++*. New York, NY: John Wiley & Sons, 1990.

Weiskamp, Keith and Lorem Heiny. *Power Graphics Using Turbo C++*. New York, NY: John Wiley & Sons, 1990.

Young, Michael. *Systems Programming in Turbo C*. Alameda, CA: Sybex, Inc, 1988.

## Object-oriented programming in general

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.

## Programming Windows applications

International Business Machines Corporation. *Systems Application Architecture: Common User Access Advanced Interface Design Guide*, IBM, 1989.

Microsoft Corporation. *Microsoft Windows User's Guide*. Redmond, WA: Microsoft Corporation, 1990.

Microsoft Corporation. *Microsoft Windows Software Development Kit: Programmer's Reference*. Redmond, WA: Microsoft Corporation, 1990.

Petzold, Charles. *Programming Windows*. Redmond, WA: Microsoft Press, 1988.

## Reference

American National Standard for Information Systems (ANSI). *Programming Language C*. Document number X3J11/90-013. Washington, DC: Computer & Business Equipment Manufacturers Association, 1990.

Barkakati, Nabajyoti. *The Waite Group's Essential Guide to Turbo C*. Indianapolis, IA: Howard W. Sams & Co, 1989.

Barkakati, Nabajyoti. *The Waite Group's Turbo C++ Bible.* Indianapolis, IA: Howard W. Sams & Co, 1990.

Bloom, Eric. *The Turbo C++ Trilogy.* Blue Ridge Summit, PA: Tab Books, Inc, 1990.

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

## C

## D

## E



## F

## G



## H



## I

## K



## L



## M

## N

## O



## P

## Q



## R



## S

## T



## U



## V



## W

2.0

GETTING STARTED

# BORLAND® C++

BORLAND

CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300.
OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, SWEDEN AND THE UNITED KINGDOM ■ PART # 14MN-TCP02 ■ BOR 1971