Chapter 3

Objects

CHAPTER GOALS

- ▶ To become familiar with objects
- ▶ To learn about the properties of several sample classes that were designed for this book
- ▶ To be able to construct objects and supply initial values
- ▶ To understand member functions and the dot notation
- ▶ To be able to modify and query the state of an object through member functions
- ▶ To write simple graphics programs containing points, lines, circles, and text (optional)

You have learned about the basic data types of C++: numbers and strings. While it is possible to write interesting programs using only numbers and strings, most useful programs need to manipulate data items that are more complex and more closely represent entities in the real world. Examples of these data items are employee records or graphical shapes.

The C++ language is ideally suited for designing and manipulating such data items, or, as they are usually called, *objects*. It requires a certain degree of technical mastery to design new object types, but it is quite easy to manipulate object types that have been designed by others. Therefore, you will first learn how to use objects that were specifically designed for use with this textbook. In Chapter 6 you will learn how to define these and other objects. Some of the most interesting data structures that we consider are from the realm of graphics. In this chapter you will learn how to use objects that let you draw graphical shapes on the computer screen.

To keep programming simple, we introduce only a few building blocks. You will find that the ability to draw simple graphics makes programming much more fun. However, the use of the graphics library is entirely optional. The remainder of this book does not depend on graphics.

CHAPTER CONTENTS

3.1 Constructing Objects

An *object* is a value that can be created, stored, and manipulated in a programming language. In that sense, the string `"Hello"` is an object. You can create it simply by using the C++ string notation `"Hello"`. You can store it in a variable like this:

```
string greeting = "Hello";
```

You can manipulate it, for example, by computing a substring:

```
cout << greeting.substr(0, 4);
```

This particular manipulation does not affect the object. After the substring is computed, the original string is unchanged. You will see object manipulations that do change objects later in this chapter.

In C++ every object must belong to a *class*. A class is a data type, just like `int` or `double`. However, classes are *programmer-defined*, whereas `int` and `double` are defined by the designers of the C++ language. At this point, you won't yet learn how to define your own classes, so the distinction between the built-in types and programmer-defined class types is not yet important.

In this chapter you will learn to work with the class `Time`, the class `Employee`, and four classes that represent graphical shapes. These classes are not part of standard C++; they have been created for use in this book.

To use the `Time` class, you must include the file `ccc_time.h`. Unlike the `iostream` or `cmath` headers, this file is not part of the standard C++ headers. Instead, the `Time` class is supplied with this book to illustrate simple objects. Because the `ccc_time.h` file is not a system header, you do not use angle brackets `< >` in the `#include` directive; instead, you use quotation marks:

```
#include "ccc_time.h"
```

The CCC prefix is another reminder that this header file is specific to the book; CCC stands for *Computing Concepts with C++ Essentials*. The online documentation of the code library that accompanies this book gives more instructions on how to add the code for the CCC objects to your program.

Suppose you want to know how many seconds will elapse between now and midnight. This sounds like a pain to compute by hand. However, the `Time` class makes the job easy. You will see how, in this section and the next. First, you will learn how to specify an object of type `Time`. The end of the day is 11:59 P.M. and 59 seconds. Here is a `Time` object representing that time:

```
Time(23, 59, 59)
```

You specify a `Time` object by giving three values: hours, minutes, and seconds. The hours are given in "military time": between 0 and 23 hours.

When a `Time` object is specified from three integer values such as 23, 59, 59, we say that the object is *constructed* from these values, and the values used in the construction are the *construction parameters*. In general, an object value is constructed as shown in Syntax 3.1.

You should think of a time object as an entity that is very similar to a number such as `7.5` or a string such as `"Hello"`. Just as floating-point values can be stored in `double` variables, `Time` objects can be stored in `Time` variables:

```
Time day_end = Time(23, 59, 59);
```

Think of this as the analog of

```
double interest_rate = 7.5;
```

or

```
string greeting = "Hello";
```

There is a shorthand for this very common situation (See Syntax 3.2).

```
Time day_end(23, 59, 59);
```

Syntax 3.1 : Object Construction

Class_name(*construction parameters*)

Example: `Time(19, 0, 0)`

Purpose: Construct a new object for use in an expression.

Syntax 3.2 : Object Variable Definition

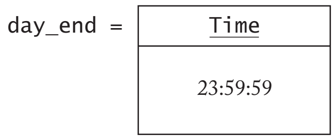
Class_name variable_name(*construction parameters*);

Example: `Time homework_due(19, 0, 0);`

Purpose: Define a new object variable and supply parameter values for initialization.

Figure 1

A Time Object



This defines a variable `day_end` that is initialized to the `Time` object `Time(23,59,59)`. (See Figure 1.)

Many classes have more than one construction mechanism. For example, there are two methods for constructing times: by specifying hours, minutes, and seconds, and by specifying no parameters at all. The expression

```
Time()
```

creates an object representing the current time, that is, the time when the object is constructed. Making an object with no construction parameter is called *default construction*.

Of course, you can store a default `Time` object in a variable:

```
Time now = Time();
```

The shorthand notation for using default construction is slightly inconsistent:

```
Time now; /* OK. This defines a variable and invokes the default constructor. */
```

and not

```
Time now(); /* NO! This does not define a variable */
```

For strange historical reasons, you cannot use `()` when defining a variable with default construction.

3.2 Using Objects

Once you have a `Time` variable, what can you do with it? Here is one useful operation. You can add a certain number of seconds to the time:

```
wake_up.add_seconds(1000);
```

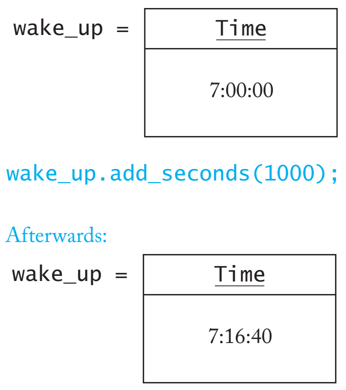
Afterwards, the object in the variable `wake_up` is changed. It is no longer the time value assigned when the object was constructed, but a time object representing a time that is exactly 1,000 seconds from the time previously stored in `wake_up`. (See Figure 2.)

Whenever you apply a function (such as `add_seconds`) to an object variable (such as `wake_up`), you use the same dot notation that we already used for certain string functions:

```
int n = greeting.length();
cout << greeting.substr(0, 4);
```

Figure 2

Changing the State of an Object



A function that is applied to an object with the dot notation is called a *member function* in C++.

Now that you've seen how to change the state of a time object, how can you find out the current time stored in the object?You have to ask it. There are three member functions for this purpose, called

```
get_seconds()
get_minutes()
get_hours()
```

They too are applied to objects using the dot notation. (See Figure 3.)

File time1.cpp

```
#include <iostream>

using namespace std;

#include "ccc_time.h"

int main()
{
   Time wake_up(7, 0, 0);
   wake_up.add_seconds(1000); /* a thousand seconds later */
   cout << wake_up.get_hours()
      << ":" << wake_up.get_minutes()
      << ":" << wake_up.get_seconds() << "\n";

   return 0;
}
```

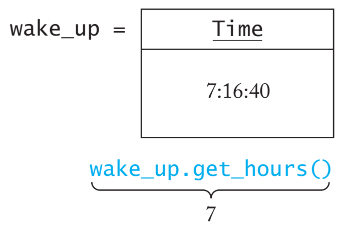


Figure 3

Querying the State of an Object

This program displays

```
7:16:40
```

Since you can *get* the hours of a time, it seems natural to suggest that you can *set* it as well:

```
homework_due.set_hours(2); /* No! Not a supported member function */
```

`Time` objects do not support this member function. There is a good reason, of course. Not all hour values make sense. For example,

```
homework_due.set_hours(9999); /* Doesn't make sense */
```

Of course, one could try to come up with some meaning for such a call, but the author of the `Time` class decided simply not to supply these member functions. Whenever you use an object, you need to find out which member functions are supplied; other operations, however useful they may be, are simply not possible.

The `Time` class has only one member function that can modify `Time` objects: `add_seconds`. For example, to advance a time by one hour, you can use

```
const int SECONDS_PER_HOUR = 60 * 60;
homework_due.add_seconds(SECONDS_PER_HOUR);
```

You can move the time back by an hour:

```
homework_due.add_seconds(-SECONDS_PER_HOUR);
```

If you are entirely unhappy with the current object stored in a variable, you can overwrite it with another one:

```
homework_due = Time(23, 59, 59);
```

Figure 4 shows this replacement.

There is one final member function that a time variable can carry out: It can figure out the number of seconds between itself and another time. For example, the following

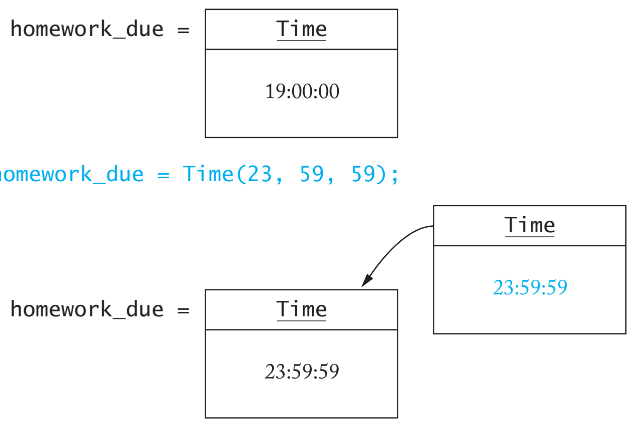


Figure 4

Replacing an Object with Another

program computes the number of seconds between the current time and the last second of the day.

File time2.cpp

```
#include <iostream>

using namespace std;

#include "ccc_time.h"

int main()
{
   Time now;
   Time day_end(23, 59, 59);
   int seconds_left = day_end.seconds_from(now);

   cout << "There are "
      << seconds_left
      << " seconds left in this day.\n";

   return 0;
}
```

To summarize, in C++ objects are constructed by writing the class name, followed by construction parameters in parentheses. There is a shortcut notation for initializing an object variable. Member functions are applied to objects and object variables with the dot notation. The functions of the `Time` class are listed in Table 1.

Name	Purpose
`Time()`	Constructs the current time
`Time(h, m, s)`	Constructs the time with hours `h`, minutes `m`, and seconds `s`
`t.get_seconds()`	Returns the seconds value of `t`
`t.get_minutes()`	Returns the minutes value of `t`
`t.get_hours()`	Returns the hours value of `t`
`t.add_seconds(n)`	Changes `t` to move by `n` seconds
`t.seconds_from(t2)`	Computes the number of seconds between `t` and `t2`

Table 1

Member Functions of the `Time` Class

Common Error 3.1

Trying to Call a Member Function without a Variable

Suppose your code contains the instruction

```
add_seconds(30); /* Error */
```

The compiler will not know which time to advance. You need to supply a variable of type `Time`:

```
Time liftoff(19, 0, 0);
liftoff.add_seconds(30);
```

Productivity Hint 3.1

Keyboard Shortcuts for Mouse Operations

Programmers spend a lot of time with the keyboard and the mouse. Programs and documentation are many pages long and require a lot of typing. The constant switching between the editor, compiler, and debugger takes up quite a few mouse clicks. The designers of programs such as a C++ integrated development environment have added some features to make your work easier, but it is up to you to discover them.

Just about every program has a user interface with menus and dialog boxes. Click on a menu and click on a submenu to select a task. Click on each field in a dialog box, fill in the requested answer, and click OK. These are great user interfaces for the beginner, because they are easy to master, but they are terrible user interfaces for the regular user. The constant switching between the keyboard and the mouse slows you down. You need to move a hand off the keyboard, locate the mouse, move the mouse, click the mouse, and move the hand back onto the keyboard. For that reason, most user interfaces have *keyboard shortcuts:* combinations of keystrokes that allow you to achieve the same tasks without having to switch to the mouse at all.

Many common applications use the following conventions:

- Pressing the Alt key plus the underlined key in a menu (as in "File") pulls down that menu. Inside a menu, just provide the underlined character in the submenu to activate it. For example, Alt+`F O` selects "File" "Open". Once your fingers know about this combination, you can open files faster than the fastest mouse artist.
- Inside dialog boxes, the Tab key is important; it moves from one option to the next. The arrow keys move within an option. The Enter key accepts all the options selected in the dialog box, and the escape key (Esc) cancels any changes.
- In a program with multiple windows, Ctrl+Tab toggles through the windows managed by that program, for example between the source and error window.
- Alt+Tab toggles between applications, letting you quickly toggle between, for example, the compiler and a folder explorer program.

- Hold down the Shift key and press the arrow keys to highlight text. Then use Ctrl+`X` to cut the text, Ctrl+`C` to copy it, and Ctrl+`V` to paste it. These keys are easy to remember. The `V` looks like an insertion mark that an editor would use to insert text. The `X` should remind you of crossing out text. The `C` is just the first letter in "Copy". (OK, so it is also the first letter in "Cut"—no mnemonic rule is perfect.) You find these reminders in the Edit menu.

Of course, the mouse has its use in text processing: to locate or select text that is on the same screen but far away from the cursor.

Take a little bit of time to learn about the keyboard shortcuts that the designers of your programs provided for you; the time investment will be repaid many times during your programming career. When you blaze through your work in the computer lab with keyboard shortcuts, you may find yourself surrounded by amazed onlookers who whisper, "I didn't know you could do *that*".

3.3 Real-Life Objects

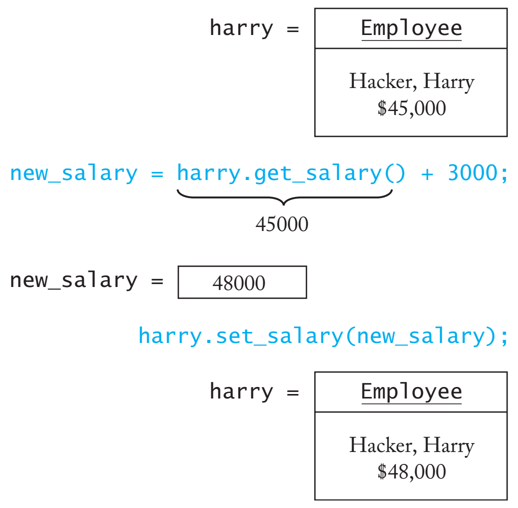
One reason for the popularity of object-oriented programming is that it is easy to *model* entities from real life in computer programs, making programs easy to understand and modify. Consider the following program:

File employee.cpp

```
#include <iostream>

using namespace std;

#include "ccc_empl.h"

int main()
{
   Employee harry("Hacker, Harry", 45000.00);

   double new_salary = harry.get_salary() + 3000;
   harry.set_salary(new_salary);

   cout << "Name: " << harry.get_name() << "\n";
   cout << "Salary: " << harry.get_salary() << "\n";

   return 0;
}
```

This program creates a variable `harry` and initializes it with an object of type `Employee`. There are two construction parameters: the name of the employee and the starting salary.

We then give Harry a $3,000 raise (see Figure 5). We first find his current salary with the `get_salary` member function. We determine the new salary by adding $3,000. We use the `set_salary` member function to set the new salary.



```
new_salary = harry.get_salary() + 3000;
harry.set_salary(new_salary);
```

Figure 5

An Employee Object

Finally, we print out the name and salary number of the employee object. We use the `get_name` and `get_salary` member functions to get the name and salary.

As you can see, this program is easy to read because it carries out its computations with meaningful entities, namely employee objects.

Note that you can change the salary of an employee with the `set_salary` member function. However, you cannot change the name of an `Employee` object.

This `Employee` class, whose functions are listed in Table 2, is not very realistic. In real data-processing programs, employees also have ID numbers, addresses, job titles, and so on. To keep the sample programs in this book simple, this class has been stripped down to the most basic properties of employees. You need to include the header file `ccc_empl.h` in all programs that use the `Employee` class.

Name	Purpose
`Employee(n, s)`	Constructs an employee with name `n` and salary `s`
`e.get_name()`	Returns the name of `e`
`e.get_salary()`	Returns the salary of `e`
`e.set_salary(s)`	Sets salary of `e` to `s`

Table 2

Member Functions of the `Employee` Class

Productivity Hint 3.2

Using the Command Line Effectively

If your programming environment allows you to accomplish all routine tasks with menus and dialog boxes, you can skip this note. However, if you need to invoke the editor, the compiler, the linker, and the program manually to test, then it is well worth learning about *command line editing*.

Most operating systems (UNIX, Macintosh OS X, Windows) have a *command line interface* to interact with the computer. (In Windows, you can use the DOS command line interface by double-clicking the “Command Prompt” icon.) You launch commands at a *prompt*. The command is executed, and upon completion you get another prompt. Most professional programmers use the command line interface for repetitive tasks because it is much faster to type commands than to navigate windows and buttons.

When you develop a program, you find yourself executing the same commands over and over. Wouldn’t it be nice if you didn’t have to type beastly commands like

```
g++ -o myprog myprog.cpp
```

more than once? Or if you could fix a mistake rather than having to retype the command in its entirety? Many command line interfaces have an option to do just that, but they don’t always make it obvious. With some versions of Windows, you need to install a program called DOSKEY. If you use UNIX, try to get the `bash` or `tcsh` shell installed for you—ask a lab assistant or system administrator to help you with the setup. With the proper setup, the up arrow key ↑ is redefined to cycle through your old commands. You can edit lines with the left and right arrow keys. You can also perform *command completion*. For example, to reissue the same `gcc` command, type `!gcc` (UNIX) or `gcc` and press F8 (Windows).

Random Fact 3.1

Mainframes—When Dinosaurs Ruled the Earth

When the International Business Machines Corporation, a successful manufacturer of punch-card equipment for tabulating data, first turned its attention to designing computers in the early 1950s, its planners assumed that there was a market for perhaps 50 such devices, for installation by the government, the military, and a few of the country’s largest corporations. Instead, they sold about 1,500 machines of their System 650 model and went on to build and sell more powerful computers.

The so-called *mainframe* computers of the fifties, sixties, and seventies were huge. They filled up a whole room, which had to be climate-controlled to protect the delicate equipment (see Figure 6). Today, because of miniaturization technology, even mainframes are getting smaller, but they are still very expensive. (At the time of this writing, the cost for a midrange IBM 3090 is approximately 4 million dollars.)

These huge and expensive systems were an immediate success when they first appeared, because they replaced many roomfuls of even more expensive employees, who had previously performed the tasks by hand. Few of these computers do any exciting computations. They

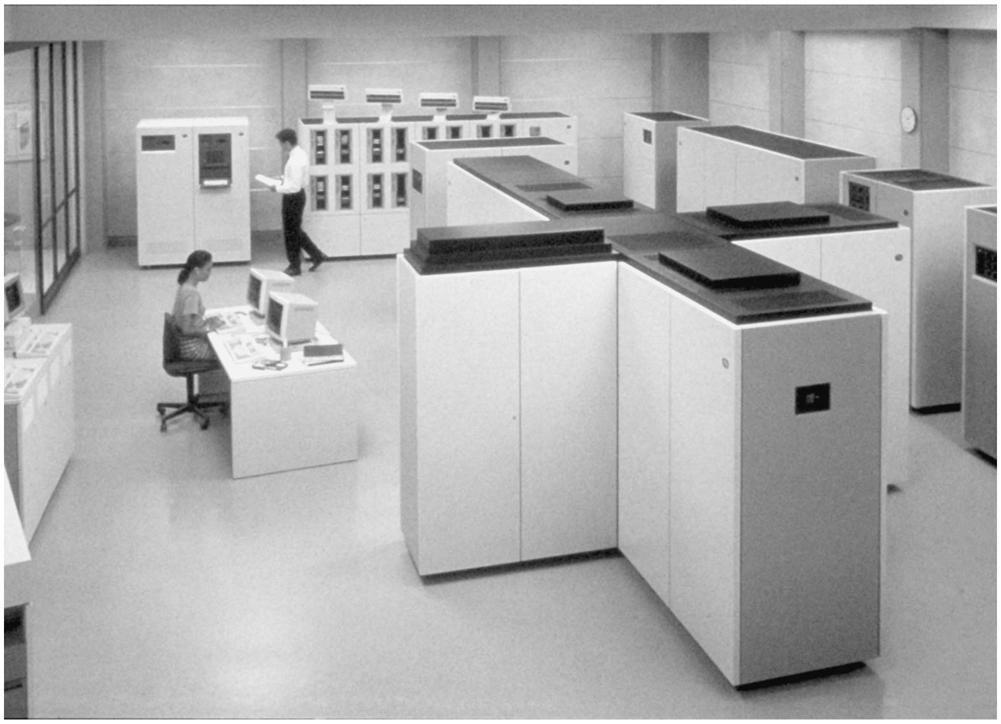


Figure 6

A Mainframe Computer

keep mundane information, such as billing records or airline reservations; the key is that they store *lots* of information.

IBM was not the first company to build mainframe computers; that honor belongs to the Univac Corporation. However, IBM soon became the major player, partially because of technical excellence and attention to customer needs, and partially because it exploited its strengths and structured its products and services in a way that made it difficult for customers to mix IBM products with those of other vendors. In the sixties its competitors, the so-called "Seven Dwarfs"—GE, RCA, Univac, Honeywell, Burroughs, Control Data, and NCR—fell on hard times. Some went out of the computer business altogether, while others tried unsuccessfully to combine their strengths by merging their computer operations. It was generally predicted that they would all eventually fail. It was in this atmosphere that the U.S. government brought an antitrust suit against IBM in 1969. The suit went to trial in 1975 and dragged on until 1982, when the Reagan Administration abandoned it, declaring it "without merit".

Of course, by then the computing landscape had changed completely. Just as the dinosaurs gave way to smaller, nimbler creatures, three new waves of computers had appeared: the minicomputers, workstations, and microcomputers, all engineered by new companies, not the Seven Dwarfs. Today, the importance of mainframes in the marketplace has diminished, and IBM, while still a large and resourceful company, no longer dominates the computer market.

Mainframes are still in use today for two reasons. They excel at handling large data volumes and, more importantly, the programs that control the business data have been refined over the last 20 or more years, fixing one problem at a time. Moving these programs to less expensive computers, with different languages and operating systems, is difficult and error-prone. Sun Microsystems, a leading manufacturer of workstations, was eager to prove that its mainframe system could be “downsized” to its own equipment. Sun eventually succeeded, but it took over five years—far longer than it expected.

3.4 Displaying Graphical Shapes

In the remainder of this chapter you will learn how to use a number of useful classes to render simple graphics. The graphics classes will provide a basis for interesting programming examples. This material is optional, and you can safely skip it if you are not interested in writing programs that draw graphical shapes.

There are two kinds of C++ programs that you will write in this course: *console applications* and *graphics applications*. Console applications read input from the keyboard (through `cin`) and display text output on the screen (through `cout`). Graphics programs read keystrokes and mouse clicks, and they display graphical shapes such as lines and circles, through a window object called `cwin`.

You already know how to write console programs. You include the header file `iostream` and use the `>>` and `<<` operators. To activate graphics for your programs, you must include the header file `ccc_win.h` into your program. Moreover, you need to supply the function `ccc_win_main` instead of `main` as the entry point to your program.

Unlike the `iostream` library, which is available on all C++ systems, this graphics library was created for use in this textbook. As with the `Time` and `Employee` classes, you need to add the code for the graphics objects to your programs. The online documentation for the code library describes this process.

It is slightly more complex to build a graphics program, and the `ccc_win` library does not support all computing platforms. If you prefer, you can use a text version of the graphics library that forms graphical shapes out of characters. The resulting output is not very pretty, but it is entirely sufficient for the majority of the examples in this book (see, for example, Figure 19). The online documentation of the code library describes how to select the text version of the graphics library.

To display a graphics object, you cannot just send it to `cout`:

```
Circle c;
. . .
cout << c; /* Won’t display the circle */
```

The `cout` stream displays characters on the terminal, not pixels in a window. Instead, you must send the characters to a window called `cwin`:

```
cwin << c; /* The circle will appear in the graphics window */
```

In the next section you will learn how to make objects that represent graphical shapes.

3.5 Graphics Structures

Points, circles, lines, and messages are the four graphical elements that you will use to create diagrams. A *point* has an *x*- and a *y*-coordinate. For example,

```
Point(1, 3)
```

is a point with *x*-coordinate 1 and *y*-coordinate 3. What can you do with a point? You can display it in a graphics window.

File point.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   cwin << Point(1, 3);

   return 0;
}
```

You frequently use points to make more complex graphical shapes.

```
Circle(Point(1, 3), 2.5);
```

This defines a circle whose center is the point with coordinates (1, 3) and whose radius is 2.5.

As always, you can store a `Point` object in a variable of type `Point`. The following code defines and initializes a `Point` variable and then displays the point. Then a circle with center `p` is created and also displayed (Figure 7).

File circle.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   Point p(1, 3);
   cwin << p << Circle(p, 2.5);

   return 0;
}
```

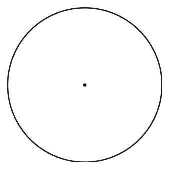


Figure 7

Output from `circle.cpp`

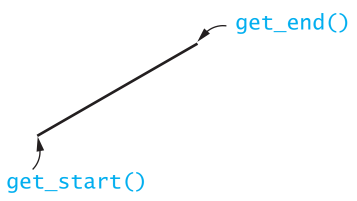


Figure 8

A Line

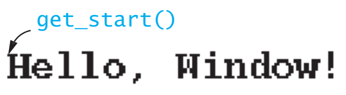


Figure 9

A Message

Two points can be joined by a *line* (Figure 8).

File line.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   Point p(1, 3);
   Point q(4, 7);
   Line s(p, q);
   cwin << s;

   return 0;
}
```

In a graphics window you can display text anywhere you like. You need to specify what you want to show and where it should appear (Figure 9).

File hellowin.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   Point p(1, 3);
   Message greeting(p, "Hello, Window!");
   cwin << greeting;

   return 0;
}
```

The point parameter specifies the *upper left corner* of the message. The second parameter can be either a string or a number.

There is one member function that all our graphical classes implement: `move`. If `obj` is a point, circle, line, or message, then

```
obj.move(dx, dy)
```

changes the position of the object, moving the entire object by `dx` units in the *x*-direction

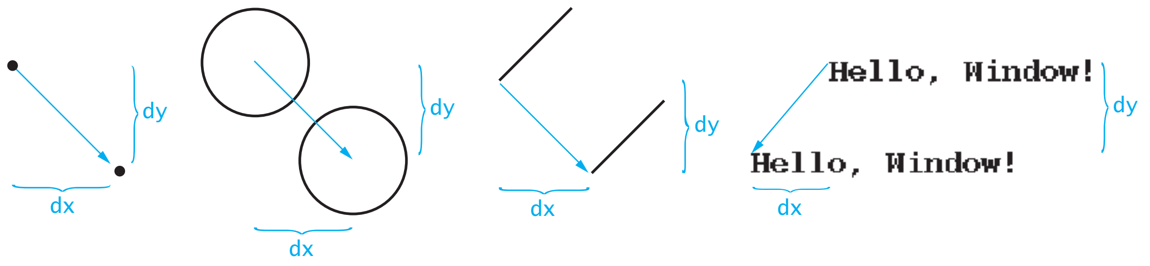


Figure 10

The `move` Operation

and `dy` units in the *y*-direction. Either or both of `dx` and `dy` can be zero or negative (see Figure 10). For example, the following code draws a square (see Figure 11).

File square.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   Point p(1, 3);
   Point q = p;
   Point r = p;
   q.move(0, 1);
   r.move(1, 0);
   Line s(p, q);
   Line t(p, r);
   cwin << s << t;
   s.move(1, 0);
   t.move(0, 1);
   cwin << s << t;

   return 0;
}
```

After a graphical object has been constructed and perhaps moved, you sometimes want to know where it is currently located. There are two member functions for `Point` objects: `get_x` and `get_y`. They get the *x*- and *y*-positions of the point.

The `get_center` and `get_radius` member functions return the center and radius of a circle. The `get_start` and `get_end` member functions return the starting point and end

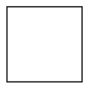


Figure 11

Square Drawn by `square.cpp`

point of a line. The `get_start` and `get_text` member functions on a `Message` object return the starting point and the message text. Since `get_center`, `get_start`, and `get_end` return `Point` objects, you may need to apply `get_x` or `get_y` to them to determine their *x*- and *y*-coordinates. For example,

```
Circle c(. . .);
. . .
double cx = c.get_center().get_x();
```

You now know how to construct graphical objects, and you have seen all member functions for manipulating and querying them (summarized in Tables 3 through 6). The design of these classes was purposefully kept simple, but as a result some common tasks require a little ingenuity (see Productivity Hint 3.3).

Table 3

Functions of the `Point` Class

Name	Purpose
`Point(x, y)`	Constructs a point at location (`x`, `y`)
`p.get_x()`	Returns the *x*-coordinate of point `p`
`p.get_y()`	Returns the *y*-coordinate of point `p`
`p.move(dx, dy)`	Moves point `p` by (`dx`, `dy`)

Table 4

Functions of the `Circle` Class

Name	Purpose
`Circle(p, r)`	Constructs a circle with center `p` and radius `r`
`c.get_center()`	Returns the center point of circle `c`
`c.get_radius()`	Returns the radius of circle `c`
`c.move(dx, dy)`	Moves circle `c` by (`dx`, `dy`)

Table 5

Functions of the `Line` Class

Name	Purpose
`Line(p, q)`	Constructs a line joining points `p` and `q`
`l.get_start()`	Returns the starting point of line `l`
`l.get_end()`	Returns the ending point of line `l`
`l.move(dx, dy)`	Moves line `l` by (`dx`, `dy`)

Table 6

Functions of the `Message` Class

Name	Purpose
`Message(p, s)`	Constructs a message with starting point `p` and text string `s`
`Message(p, x)`	Constructs a message with starting point `p` and a label equal to the number `x`
`m.get_start()`	Returns the starting point of message `m`
`m.get_text()`	Gets the text string of message `m`
`m.move(dx, dy)`	Moves message `m` by (`dx`, `dy`)

Productivity Hint 3.3

Think of Points as Objects, Not Pairs of Numbers

Suppose you want to draw a square starting with the point p as the upper left corner and with side length 1. If p has coordinates (p_x, p_y), then the upper right corner is the point with coordinates $(p_x + 1, p_y)$. Of course, you can program that:

```
Point q(p.get_x() + 1, p.get_y()); /* Cumbersome */
```

Try to think about points as objects, not pairs of numbers. Taking this point of view, there is a more elegant solution: Initialize `q` to be the same point as `p`, then move it to where it belongs:

```
Point q = p;
q.move(1, 0); /* Simple */
```

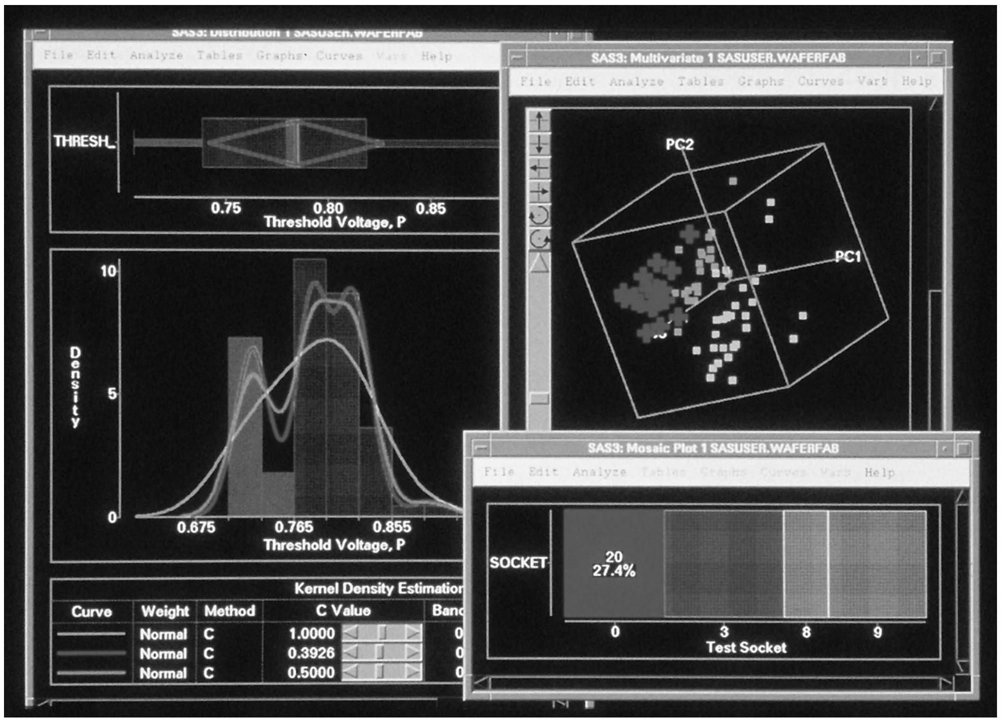
Random Fact 3.2

Computer Graphics

The generation and manipulation of visual images is one of the most exciting applications of the computer. We distinguish between different kinds of graphics.

Diagrams, such as numeric charts or maps, are artifacts that convey information to the viewer (see Figure 12). They do not directly depict anything that occurs in the natural world, but are a tool for visualizing information.

Scenes are computer-generated images that attempt to depict images of the real or an imagined world (see Figure 13). It turns out to be quite a challenge to render light and shadows accurately. Special effort must be taken so that the images do not look too neat and simple; clouds, rocks, leaves, and dust in the real world have a complex and somewhat random appearance. The degree of realism in these images is constantly improving.


SAS3: Distribution 1 SASUSER.WAFERFAB
File Edit Analyze Tables Graphs Curves Vars Help
THRESH_
0.75
0.80
0.85
Threshold Voltage, P
10
5
0
Density
0.675
0.765
0.855
Threshold Voltage, P
SAS3: Multivariate 1 SASUSER.WAFERFAB
File Edit Analyze Tables Graphs Curves Vars Help
PC2
PC1
SAS3: Mosaic Plot 1 SASUSER.WAFERFAB
File Edit Analyze Tables Graphs Curves Vars Help
SOCKET
20
27.4%
0
3
8
9
Test Socket

Kernel Density Estimatio

Curve	Weight	Method	C Value	Ban
	Normal	C	1.0000	0
	Normal	C	0.3926	0
	Normal	C	0.5000	0

Figure 12

Diagrams

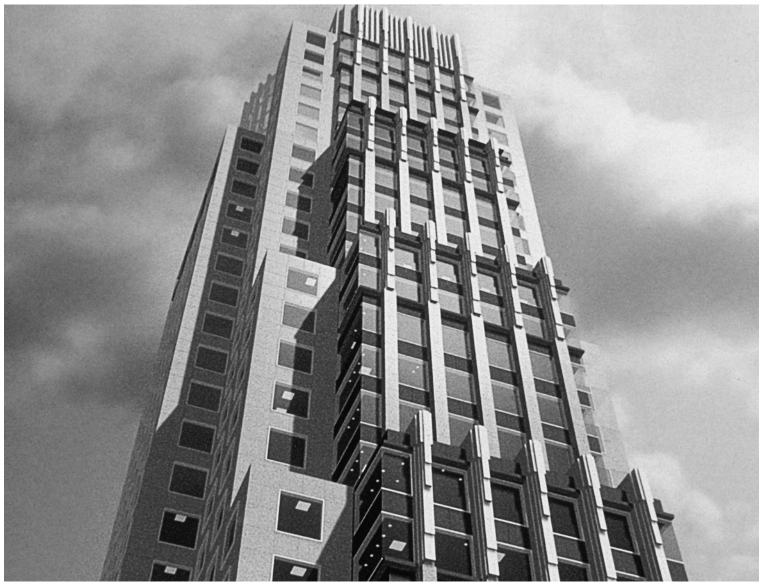


Figure 13

Scene

Figure 14

Manipulated Image

Manipulated images are photographs or film footage of actual events that have been converted to digital form and edited by the computer (see Figure 14). For example, film sequences of the movie *Apollo 13* were produced by starting from actual images and changing the perspective, showing the launch of the rocket from a more dramatic viewpoint.

Computer graphics is one of the most challenging fields in computer science. It requires processing of massive amounts of information at very high speed. New algorithms are constantly invented for this purpose. Viewing overlapping three-dimensional objects with curved boundaries requires advanced mathematical tools. Realistic modeling of textures and biological entities requires extensive knowledge of mathematics, physics, and biology.

3.6 Choosing a Coordinate System

We need to have an agreement on the meaning of particular coordinates. For example, where is the point with *x*-coordinate 1 and *y*-coordinate 3 located? Some graphics systems use pixels, the individual dots on the display, as coordinates, but different displays have different pixel counts and densities. Using pixels makes it difficult to write programs that look pleasant on every display screen. The library supplied with this book uses a coordinate system that is independent of the display.

Figure 15 shows the default coordinate system used by this book's library. The origin is at the center of the screen, and the *x*-axis and *y*-axis are 10 units long in either direction. The axes do not actually appear (unless you create them yourself by drawing `Line` objects).

This default coordinate system is fine for simple test programs, but it is *useless* when dealing with real data. For example, suppose we want to show a graph plotting the average temperature (degrees Celsius) in Phoenix, Arizona, for every month of the year. The temperature ranges from 11°C in January to 33°C in July (see Table 7).

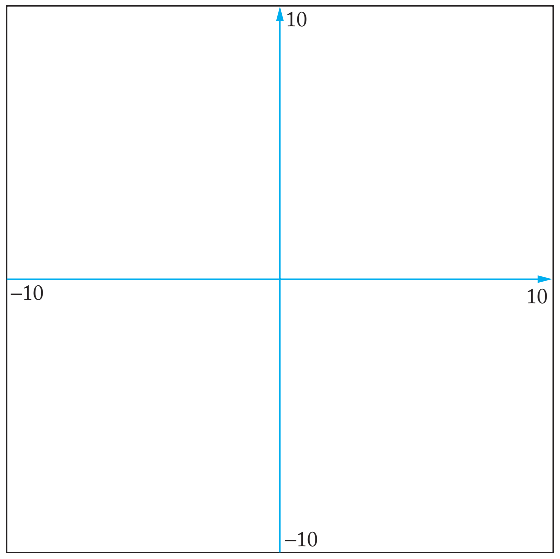


Figure 15

Default Coordinate System for Graphics Library

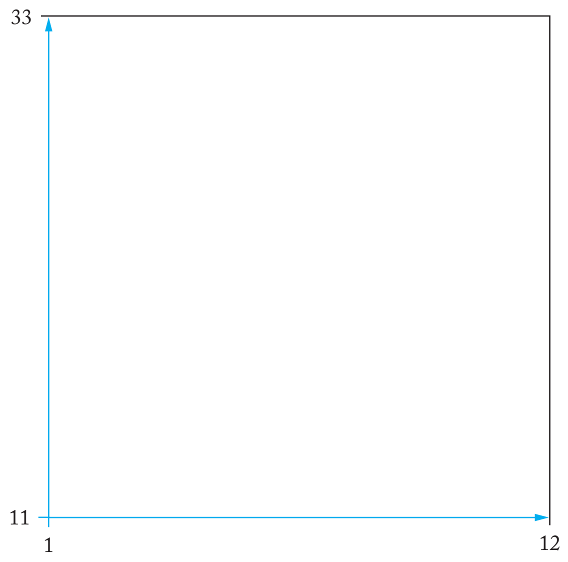


Figure 16

Coordinate System for Temperature

Even the January data

```
cwin << Point(1, 11);
```

won't show up in the window at all! In this situation, we need to change from the default coordinate system to one that makes sense for our particular program. Here, the *x*-coordinates are the month values, ranging from 1 to 12. The *y*-coordinates are the temperature values, ranging from 11 to 33. Figure 16 shows the coordinate system that we need. As you can see, the top left corner is (1, 33) and the bottom right corner is (12, 11).

Table 7

Average Temperatures in Phoenix, Arizona

Month	Average Temperature	Month	Average Temperature
January	11°C	July	33°C
February	13°C	August	32°C
March	16°C	September	29°C
April	20°C	October	23°C
May	25°C	November	16°C
June	31°C	December	12°C

To select this coordinate system, use the following instruction:

```
cwin.coord(1, 33, 12, 11);
```

Following a common convention in graphics systems, you must first specify the desired coordinates for the *top left* corner (which has *x*-coordinate 1 and *y*-coordinate 33), then the desired coordinates for the bottom right corner ($x = 12, y = 11$).

Here is the complete program:

File phoenix.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   cwin.coord(1, 33, 12, 11);
   cwin << Point(1, 11);
   cwin << Point(2, 13);
   cwin << Point(3, 16);
   cwin << Point(4, 20);
   cwin << Point(5, 25);
   cwin << Point(6, 31);
   cwin << Point(7, 33);
   cwin << Point(8, 32);
   cwin << Point(9, 29);
   cwin << Point(10, 23);
   cwin << Point(11, 16);
   cwin << Point(12, 12);

   return 0;
}
```

Figure 17 shows the output of the program.

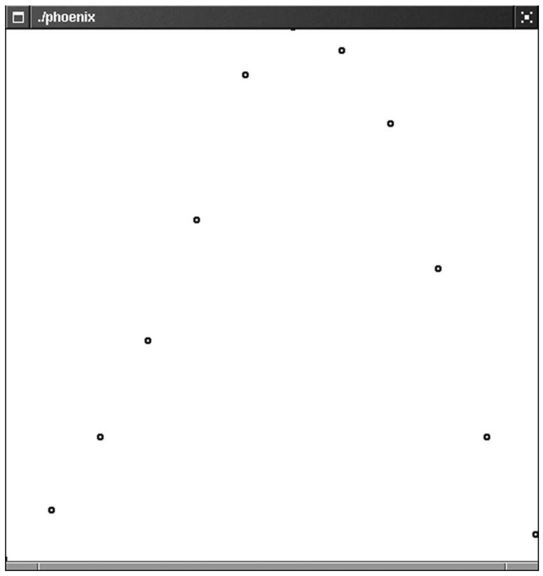


Figure 17

Average Temperatures in Phoenix, Arizona

Productivity Hint 3.4

Choose a Convenient Coordinate System

Whenever you deal with real-world data, you should set a coordinate system that is matched to the data. Figure out which range of *x*- and *y*-coordinates is most convenient for you. For example, suppose you want to display a tic-tac-toe board (see Figure 18).

You could labor mightily and figure out where the lines are in relation to the default coordnate system, or you can simply set your own coordinate system with (0, 0) in the top left corner and (3, 3) in the bottom right corner.

```
#include "ccc_win.h"

int ccc_win_main()
{
   cwin.coord(0, 0, 3, 3);
   Line horizontal(Point(0, 1), Point(3, 1));
   cwin << horizontal;
   horizontal.move(0, 1);
   cwin << horizontal;
   Line vertical(Point(1, 0), Point(1, 3));
   cwin << vertical;
   vertical.move(1, 0);
   cwin << vertical;

   return 0;
}
```

Some people have horrible memories about coordinate transformations from their high school geometry class and have taken a vow never to think about coordinates again for the remainder of their lives. If you are among them, you should reconsider. In the CCC graphics library, coordinate systems are your friend—they do all the horrible algebra for you, so you don't have to program it by hand.

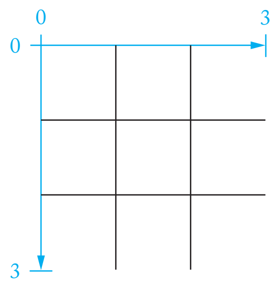


Figure 18

Coordinate System for a Tic-Tac-Toe Board

3.7 Getting Input from the Graphics Window

Just as stream output does not work with the graphics window, you cannot use stream input either. Instead, you must ask the window to get input for you. The command is

```
string response = cwin.get_string(prompt);
```

This is how you inquire about the user name:

```
string name = cwin.get_string("Please type your name:");
```

The prompt and a field for typing the input are displayed in a special input area. Depending on your computer system, the input area is in a dialog box or at the top or bottom of the graphics window. The user can then type input. After the user hits the Enter key, the user's keystrokes are placed into the `name` string. The message prompt is then removed from the screen.

The `get_string` function always returns a string. Use `get_int` or `get_double` to read an integer or floating-point number:

```
int age = cwin.get_int("Please enter your age:");
```

The user can specify a point with the mouse. To prompt the user for mouse input, use

```
Point response = cwin.get_mouse(prompt);
```

Name	Purpose
`w.coord(x1, y1, x2, y2)`	Sets the coordinate system for subsequent drawing: (`x1`, `y1`) is the top left corner, (`x2`, `y2`) the bottom right corner
`w << x`	Displays the object `x` (a point, circle, line, or message) in window `w`
`w.clear()`	Clears window `w` (erases its contents)
`w.get_string(p)`	Displays prompt `p` in window `w` and returns the entered string
`w.get_int(p)`	Displays prompt `p` in window `w` and returns the entered integer
`w.get_double(p)`	Displays prompt `p` in window `w` and returns the entered floating-point value
`w.get_mouse(p)`	Displays prompt `p` in window `w` and returns the mouse click point

Table 8

Functions of the `GraphicWindow` Class

For example,

```
Point center = cwin.get_mouse("Click center of circle");
```

The user can move the mouse to the desired location. Once the user clicks on the mouse button, the prompt is cleared and the selected point is returned.

Here is a program that puts these functions (summarized in Table 8) to work. It asks the user to enter a name and to try to click inside a circle. Then the program displays the point that the user specified.

File click.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   string name = cwin.get_string("Please type your name:");
   Circle c(Point(0, 0), 1);
   cwin << c;
   Point m = cwin.get_mouse("Please click inside the circle.");
   cwin << m << Message(m, name + ", you clicked here");

   return 0;
}
```

3.8 Comparing Visual and Numerical Information

The next example shows how one can look at the same problem both visually and numerically. You want to determine the intersection between a line and a circle. The circle is centered on the screen. The user specifies a radius of the circle and the *y*-intercept of a horizontal line.You then draw the circle and the line.

File intsect1.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   double radius = cwin.get_double("Radius: ");
   Circle c(Point(0, 0), radius);

   double b = cwin.get_double("Line position: ");
   Line s(Point(-10, b), Point(10, b));

   cwin << c << s;

   return 0;
}
```

Figure 19 shows the output of this program.

```
                       ***************
                 ******                    ******
              ***                               ***
**************************************************************
           **                                   **
        **                                         **
       *                                             *
      *                                               *
     *                                                 *
     *                                                 *
     *                                                 *
      *                                               *
       *                                             *
        **                                         **
          **                                     **
            **                                 **
              ***                           ***
                 ******                    ******
                       ***************
```

Figure 19

Intersection of a Line and a Circle
(Using the Text Version of the Graphics Library)

Now suppose you would like to know the *exact* coordinates of the intersection points. The equation of the circle is

$$x^2 + y^2 = r^2$$

where r is the radius (which was given by the user). You also know y. A horizontal line has equation $y = b$, and b is another user input. Thus x is the remaining unknown, and you can solve for it. You expect two solutions, corresponding to

$$x_{1,2} = \pm\sqrt{r^2 - b^2}$$

Plot both points and label them with the numerical values. If you do it right, these two points will show up right on top of the actual intersections in the picture. If you do it wrong, the two points will be at the wrong place.

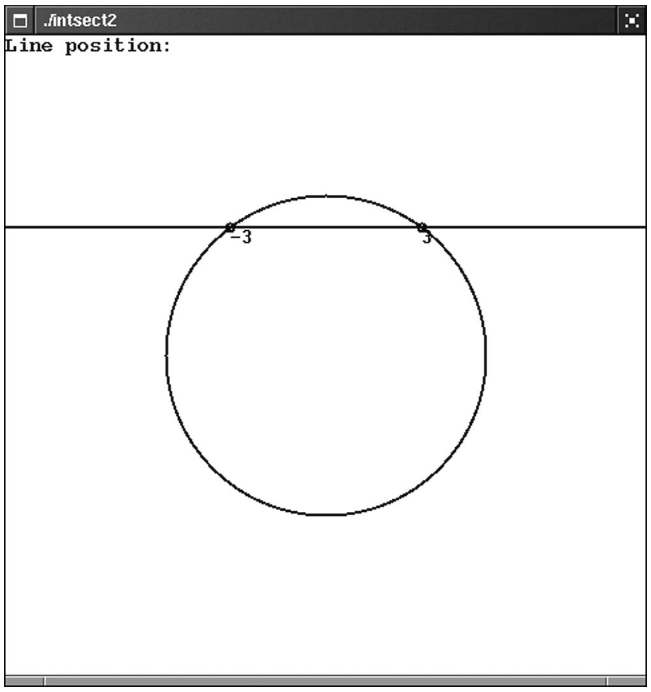
Here is the code to compute and plot the intersection points.

File intsect2.cpp

```
#include "ccc_win.h"

int ccc_win_main()
{
   double radius = cwin.get_double("Radius: ");
   Circle c(Point(0, 0), radius);

```

```
    double b = cwin.get_double("Line position: ");
    Line s(Point(-10, b), Point(10, b));

    cwin << c << s;

    double root = sqrt(radius * radius - b * b);

    Point p1(root, b);
    Point p2(-root, b);

    Message m1(p1, p1.get_x());
    Message m2(p2, p2.get_x());

    cwin << p1 << p2 << m1 << m2;

    return 0;
}
```

Figure 20 shows the combined output. The results match perfectly, so you can be confident that you did everything correctly. See Quality Tip 3.1 for more information on verifying that this program works correctly.



```
Line position:
```

Figure 20

Computing the Intersection Points

At this point you should be careful to specify only lines that intersect the circle. If the line doesn't meet the circle, then the program will attempt to compute a square root of a negative number, and it will terminate with a math error. You do not yet know how to implement a test to protect against this situation. That will be the topic of the next chapter.

Quality Tip 3.1

Calculate Sample Data Manually

It is difficult or impossible to prove that a given program functions correctly in all cases. For gaining confidence in the correctness of a program, or for understanding why it does not function as it should, manually calculated sample data are invaluable. If the program arrives at the same results as the manual calculation, your confidence in it is strengthened. If the manual results differ from the program results, you have a starting point for the debugging process.

Surprisingly, many programmers are reluctant to perform any manual calculations as soon as a program carries out the slightest bit of algebra. Their math phobia kicks in, and they irrationally hope that they can avoid the algebra and beat the program into submission by random tinkering, such as rearranging the + and – signs. Random tinkering is always a great time sink, but it rarely leads to useful results.

It is much smarter to look for test cases that are easy to compute and representative of the problem to be solved. The example in Figure 21 shows three easy cases that can be computed by hand and then compared against program runs.

First, let the horizontal line pass through the center of the circle. Then you expect the distance between the center and the intersection point to be the same as the radius of the circle. Let the radius be 2. The *y* position is 0 (the center of the window). You expect

$$x_1 = \sqrt{2^2 - 0^2} = 2, \quad x_2 = -2$$

Now, that wasn't so hard.

Next, let the horizontal line touch the circle on the top. Again, fix the radius to be 2. Then the *y* position is also 2, and of course $x_1 = x_2 = 0$. That was pretty easy, too.

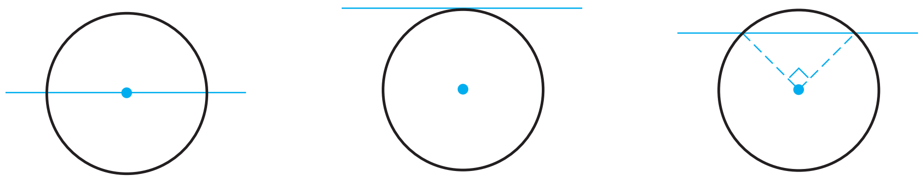


Figure 21

Three Test Cases

The first two cases were *boundary test cases* of the problem. A program may work correctly for several special cases but still fail for more typical input values. Therefore you must come up with an intermediate test case, even if it means a bit more computation. Choose a configuration where the center of the circle and the points of intersection form a right triangle. If the radius of the circle is again 2, then the height of the triangle is $\frac{1}{2}\sqrt{2}$. This looks complicated; try instead choosing the height of the triangle to be 2. Thus, the base has length 4, and the radius of the circle is $2\sqrt{2}$. Therefore, enter radius 2.828427, enter *y*-position 2, and expect $x_1 = 2, x_2 = -2$.

Running the program with these three inputs confirms the manual calculations. The computer calculations and the manual reasoning did not use the same formulas, so you can have a great deal of confidence in the validity of the program.

Random Fact 3.3

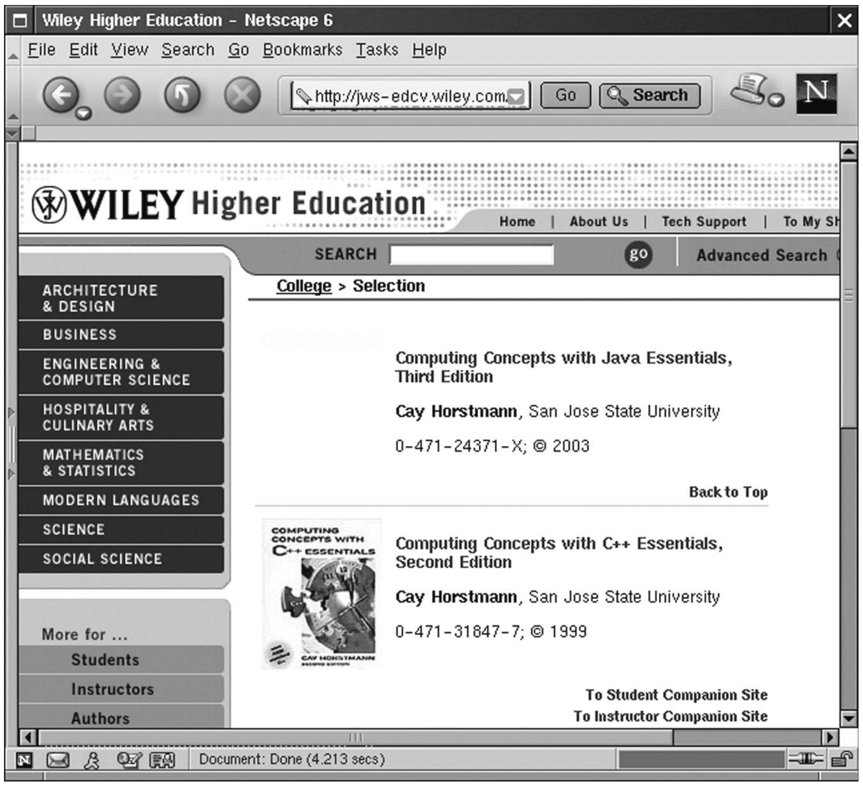
Computer Networks and the Internet

Home computers and laptops are usually self-contained units with no permanent connection to other computers. Office and lab computers, however, are usually connected with each other and with larger computers: so-called *servers*. A server can store application programs and make them available on all computers on the network. Servers can also store data, such as schedules and mail messages, that everyone can retrieve. Networks that connect the computers in one building are called *local area networks*, or LANs.

Other networks connect computers in geographically dispersed locations. Such networks are called *wide area networks* or WANs. The most prominent wide area network is the *Internet*. At the time of this writing, the Internet is in a phase of explosive growth. In 1994 the Internet connected about two million computers. Nobody knows for certain how many users have access to the Internet, but in 2002 the user population is estimated to be about half a billion. The Internet grew out of the ARPAnet, a network of computers at universities that was funded by the Advanced Research Planning Agency of the U.S. Department of Defense. The original motivation behind the creation of the network was the desire to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent, though, that remote execution was not what the network was actually used for. The principal usage was *electronic mail:* the transfer of messages between computer users at different locations. To this day, electronic mail is one of the most compelling applications of the Internet.

Over time, more and more *information* became available on the Internet. The information was created by researchers and hobbyists and made freely available to anyone, either out the goodness of their hearts or for self-promotion. For example, the GNU project is producing a set of high-quality operating system utilities and program development tools that can be used freely by anyone (`ftp://prep.ai.mit.edu/pub/gnu`). Project Gutenberg makes available the text of important classic books, whose copyright has expired, in computer-readable form (`http://www.promo.net/pg`).

The first interfaces to retrieve this information were clumsy and hard to use. All that changed with the appearance of the *World Wide Web* (WWW). The World Wide Web brought two major advances to Internet information. The information could contain *graphics* and *fonts*—a great improvement over the older text-only format—and it became possible to


Wiley Higher Education - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://jws-edcv.wiley.com
Go
Search
WILEY Higher Education
Home | About Us | Tech Support | To My Sh
SEARCH
go
Advanced Search
ARCHITECTURE & DESIGN
BUSINESS
ENGINEERING & COMPUTER SCIENCE
HOSPITALITY & CULINARY ARTS
MATHEMATICS & STATISTICS
MODERN LANGUAGES
SCIENCE
SOCIAL SCIENCE
More for ...
Students
Instructors
Authors
Back to Top
To Student Companion Site
To Instructor Companion Site
Document: Done (4.213 secs)

College > Selection

Computing Concepts with Java Essentials, Third Edition

Cay Horstmann, San Jose State University

0-471-24371-X; © 2003

Computing Concepts with C++ Essentials, Second Edition

Cay Horstmann, San Jose State University

0-471-31847-7; © 1999

Figure 22

A Web Browser

embed *links* to other information pages. Using a *browser* such as *Netscape*, exploring the information becomes easy and fun (Figure 22).

Chapter Summary

1. We use objects in programs when we need to manipulate data that are more complex than just numbers and strings. Every object belongs to a class. A class determines the behavior of its objects. In this chapter you became familiar with objects from a number of classes that were predefined for use with this textbook. However, you must wait until Chapter 6 to be able to define your own classes.
2. Objects are constructed with the constructor notation. Once an object is constructed, member functions can be applied to it with the dot notation.
3. This book describes a library of graphical structures that are used for interesting and entertaining examples. Points, lines, circles, and messages can be displayed in a win-

dow on the computer screen. Programs can obtain both text and mouse input from the user. When writing programs that display data sets, you should select a coordinate system that fits the data points.

Further Reading

[1] C. Eames and R. Eames, *A Computer Perspective*, Harvard Press, Cambridge, MA, 1973. A pictorial based on an exhibition of the history and social impact of computing. It contains many entertaining and interesting pictures of historic computing devices, their inventors, and their impact on modern life.

Review Exercises

Exercise R3.1. Explain the difference between an object and a class.

Exercise R3.2. Give the C++ code for an *object* of class `Time` and for an *object variable* of class `Time`.

Exercise R3.3. Explain the differences between a member function and a non-member function.

Exercise R3.4. Explain the difference between

```
Point(3, 4);
```

and

```
Point p(3, 4);
```

Exercise R3.5. What are the construction parameters for a `Circle` object?

Exercise R3.6. What is default construction?

Exercise R3.7. Give the C++ code to construct the following objects:

(a) Lunch time
(b) The current time
(c) The top right corner of the graphics window in the default coordinate system
(d) Your instructor as an employee (make a guess for the salary)
(e) A circle filling the entire graphics window in the default coordinate system
(f) A line representing the *x*-axis from –10 to 10.

Write the code for objects, not object variables.

Exercise R3.8. Repeat the preceding exercise, but now define variables that are initialized with the required values.

Exercise R3.9. Find the errors in the following statements:

(a) `Time now();`

(b) `Point p = (3, 4);`

(c) `p.set_x(-1);`

(d) `cout << Time`

(e) `Time due_date(2004, 4, 15);`

(f) `due_date.move(2, 12);`

(g) `seconds_from(millennium);`

(h) `Employee harry("Hacker", "Harry", 35000);`

(i) `harry.set_name("Hacker, Harriet");`

Exercise R3.10. Describe all constructors of the `Time` class. List all member functions that can be used to change a `Time` object. List all member functions that don't change the `Time` object.

Exercise R3.11. What is the value of `t` after the following operations?

```
Time t;
t = Time(20, 0, 0);
t.add_seconds(1000);
t.add_seconds(-400);
```

Exercise R3.12. If `t1` and `t2` are objects of class `Time`, is the following true or false?

`t1.add_seconds(t2.seconds_from(t1))` is the same time as `t2`

Exercise R3.13. Which five classes are used in this book for graphics programming?

Exercise R3.14. What is the value of `c.get_center` and `c.get_radius` after the following operations?

```
Circle c(Point(1, 2), 3);
c.move(4, 5);
```

Exercise R3.15. You want to plot a bar chart showing the grade distribution of all students in your class (where A = 4.0, F = 0). What coordinate system would you choose to make the plotting as simple as possible?

Exercise R3.16. Let `c` be any circle. Write C++ code to plot the circle `c` and another circle that touches `c`. *Hint:* Use `move`.

Exercise R3.17. Write C++ instructions to display the letters X and T in a graphics window, by plotting line segments.

Exercise R3.18. Suppose you run the program `intsect2.cpp` and give a value of 5 for the radius of the circle and 4 for the line position. Without actually running the program, determine what values you will obtain for the intersection points.

Exercise R3.19. Introduce an error in the program `intsect2.cpp`, by computing `root = sqrt(radius * radius + b * b)`. Run the program. What happens to the intersection points?

Programming Exercises

Exercise P3.1. Write a program that asks for the due date of the next assignment (hour, minutes). Then print the number of minutes between the current time and the due date.

Exercise P3.2. Write a graphics program that prompts the user to click on three points. Then draw a triangle joining the three points. *Hint:* To give the user feedback about the click, it is a nice touch to draw the point after each click.

```
Point p = cwin.get_mouse("Please click on the first point");
cwin << p; /* Feedback for the user */
```

Exercise P3.3. Write a graphics program that prompts the user to click on the center of a circle, then on one of the points on the boundary of the circle. Draw the circle that the user specified. *Hint:* The radius of the circle is the distance between the two points, which is computed as

$$\sqrt{\left(a_x - b_x\right)^2 + \left(a_y - b_y\right)^2}$$

Exercise P3.4. Write a graphics program that prompts the user to click on two points. Then draw a line joining the points and write a message displaying the *slope* of the line; that is, the "rise over run" ratio. The message should be displayed at the *midpoint* of the line.

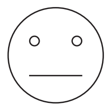
Exercise P3.5. Write a graphics program that prompts the user to click on two points. Then draw a line joining the points and write a message displaying the *length* of the line, as computed by the Pythagorean formula. The message should be displayed at the *mid-point* of the line.

Exercise P3.6. Write a graphics program that prompts the user to click on three points. Then draw a circle passing through the three points.

Exercise P3.7. Write a program that prompts the user for the first name and last name of an employee and a starting salary. Then give the employee a 5 percent raise, and print out the name and salary information stored in the employee object.

Exercise P3.8. Write a program that prompts the user for the names and salaries of three employees. Then print out the average salaries of the employees.

Exercise P3.9. Write a program to plot the following face.



Exercise P3.10. Write a program to plot the string "HELLO", using just lines and circles. Do not use the `Message` class, and do not use `cout`.

Exercise P3.11. Write a program that lets a user select two lines by prompting the user to click on both end points of the first segment, then on both end points of the second segment. Then compute the point of intersection of the lines extending through those segments, and plot it. (If the segments are parallel, then the lines don't intersect, or they are identical. In the formulas computing the intersection, this will manifest itself as a division by 0. Since you don't yet know how to write code involving decisions, your program will terminate when the division by 0 happens. Doing so is acceptable for *this* assignment.)

Here is the mathematics to compute the point of intersection. If $a = (a_x, a_y)$ and $b = (b_x, b_y)$ are the end points of the first line segment, then $ta + (1 - t)b$ runs through all points on the first line as t runs from $-\infty$ to ∞. If $c = (c_x, c_y)$ and $d = (d_x, d_y)$ are the end points of the second line segment, the second line is the collection of points $uc + (1 - u)d$. The point of intersection is the point lying on both lines. That is, it is the solution of both

$$ta + (1 - t)b = uc + (1 - u)d$$

and

$$(a - b)t + (d - c)u = d - b$$

Writing the x and y coordinates separately, we get a system of two linear equations

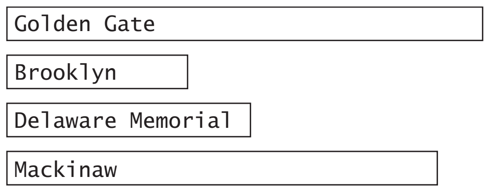
$$\begin{aligned}(a_x - b_x)t + (d_x - c_x)u &= d_x - b_x \\ (a_y - b_y)t + (d_y - c_y)u &= d_y - b_y\end{aligned}$$

Find the solutions of this system. You just need the value for t. Then compute the point of intersection as $ta + (1 - t)b$.

Exercise P3.12. *Plotting a data set.* Make a bar chart to plot a data set like the following:

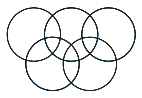
Name	Longest span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinaw	3,800

Prompt the user to type in four names and measurements. Then display a bar graph. Make the bars horizontal for easier labeling.

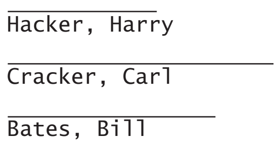


Hint: Set the window coordinates to 5,000 in the *x*-direction and 4 in the *y*-direction.

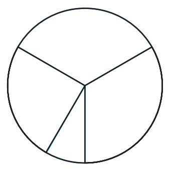
Exercise P3.13. Write a program that displays the Olympic rings. *Hint:* Construct and display the first circle, then call `move` four times.



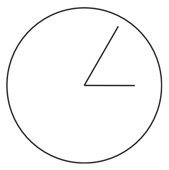
Exercise P3.14. Write a graphics program that asks the user to enter the names of three employees and their salaries. Make three employee objects. Draw a stick chart showing the names and salaries of the employees.



Exercise P3.15. Write a graphics program that asks the user to enter four data values. Then draw a pie chart showing the data values.



Exercise P3.16. Write a graphics program that draws a clock face with the current time:



Hint: You need to determine the angles of the hour hand and the minute hand. The angle of the minute hand is easy: The minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12×60 minutes.

Exercise P3.17. Write a program that tests how fast a user can type. Get the time. Ask the user to type “The quick brown fox jumps over the lazy dog”. Read a line of input. Get the current time again in another variable of type `Time`. Print out the seconds between the two times.

Exercise P3.18. Your boss, Juliet Jones, is getting married and decides to change her name. Complete the following program so that you can type in the new name for the boss:

```
int main()
{
   Employee boss("Jones, Juliet", 45000.00);
   /* your code goes here; leave the code above and below unchanged */

   cout << "Name: " << boss.get_name() << "\n";
   cout << "Salary: " << boss.get_salary() << "\n";

   return 0;
}
```

The problem is that there is no `set_name` member function for the `Employee` class. *Hint:* Make a new object of type `Employee` with the new name and the same salary. Then assign the new object to `boss`.

Exercise P3.19. Write a program that draws the picture of a house. It could be as simple as the figure below, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever).