

Chapter 1 Libraries and Interfaces: A simple

Graphics Library

Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what details one can do without and yet preserve the spirit of the whole.

— Willa Cather, **On the Art of Fiction, 1920**

OBJECTIVES

- ◆ To understand the meaning of the terms interface, package, abstraction, implementor, and client as they apply to libraries.
- ◆ To recognize that interfaces are represented in C using header files.
- ◆ To be able to read the `graphics.h` interface, which provides access to a library for drawing simple pictures on the screen, and to understand the conceptual abstraction used by the graphics library.
- ◆ To learn how to draw lines using `Movepen` and `DrawLine` and to draw arcs using `DrawArc`.
- ◆ To learn how to extend the basic capabilities of the graphics library by defining new higher-level functions.
- ◆ To practice writing large programs using the graphics package.
- ◆ To appreciate the importance of general tools and the associated strategy of bottom-up implementation.

Every program you have seen or written in this text has called at least one library function. Even the one-line program `hello.c` calls `printf` to display its message on the screen. In modern programming, it is impossible to write interesting programs without calling library functions, and by this point, you should be reasonably adept at calling them.

So far, the functions you have written have all been part of a single program. You can call them from the main program or from other functions that are part of the same program file, but you have not been able to take your own functions and put them into a library that you can then use for any number of later programs. Before you can do so, you need to learn more about what a library is and how it works. Enormous power comes from being able to design good libraries and use them well. A key part of that power comes from understanding the concept of an *interface*, which is the main topic not only of this chapter but of the next several chapters as well.

To give you a sense of what an interface is and how it works, this chapter concentrates on having you read through an existing interface rather than having you design one from scratch. Before attempting to write novels, authors usually spend many years reading them. In doing so, they learn about the form of the novel and develop their own appreciation of what makes a particular novel good. Here, your job is to learn the

basic structure of interfaces, and the best way to do that is to study existing examples. You will have the chance to write your own interfaces beginning in Chapter 8.

This chapter begins by outlining the conceptual structure of interfaces and defining several terms that make it easier to talk about them. It then introduces a simple graphics library that enables you to draw pictures on the screen. That library will make it possible for you to write programs that are much more exciting than those you have seen in the preceding chapters, which makes the graphics library interesting in its own right. As you use the graphics library, however, it's important to keep in mind such issues as how libraries work, what they contain, and how interfaces are used to describe them.

1-1 The concept of an interface

In English, the word interface means a common boundary between two distinct entities. The surface of a pond, for example, is the interface between the water and the air. In programming, an interface constitutes a conceptual boundary rather than a physical one: an **interface** is the boundary between the implementation of a library and programs that use that library. Information passes across that boundary whenever functions in that library are called. The interface mediates and gives structure to the exchange of information between the library and its users. Conceptually, a programming interface also represents a shared understanding about the nature of that boundary, providing both creators and users of a library with the critical information they need to know.

Consider, for example, the math library introduced in Chapter 5. The math library defines several functions, such as `sqrt`. Programs that use the math library can call `sqrt` to calculate a square root without having to specify the actual steps involved. Those steps are part of the implementation of the square root function, which was written by the programmers who created the math library itself. Chapter 6 presented two possible strategies—Newton's method and Taylor series expansion—for implementing the `sqrt` function. The library implementors might have sued one of those strategies or any other algorithm that computes the correct result.

Knowing how to call the `sqrt` function and knowing how to implement it are both important skills. It is critical to realize, however, that those two skills—calling a function and implementing one—are to a large extent unrelated. Successful programmers often use functions that they wouldn't know how to write. Conversely, programmers who implement a library function cannot anticipate all the potential uses for that function.

To emphasize the difference in perspective between programmers who implement a library and those who use it, computer scientists have adopted specific terms to refer to programmers working in each of these capacities. Naturally enough, a programmer who implements a library is called an **implementor**. Because the word *user* means someone who runs a program rather than someone who writes part of one, a programmer who calls functions provided by a library is called a **client** of that library¹.

¹ In computer science, the term client sometimes refers to code that uses a library and sometimes to the programmer who writes that code. If there is a possibility for confusion, I will refer to code that uses a library as client code, although I will also follow the standard convention and use client in such case, as long as the intent is clear.

Even though clients and implementors have different perspectives on the library, both must understand certain aspects of that library's design. As a client, you don't need to know the details of its operation, but you do need to know exactly how to call it. As an implementor, on the other hand, you are not directly make it possible for them to do so by providing the information they need to call the functions it contains. For each function in the library, the client must know the following:

- Its name
- The arguments it requires and the types of those arguments
- The type of result it returns

That this information is precisely what a function prototype provides is by no means a coincidence. In C, the prototype for a function and its implementation are separated because they convey information to different audiences. The client and the implementor must agree on the function prototype, which means that it is part of the interface. By contrast, only the implementor is concerned with the function implementation. The act of making a function available to clients by including its prototype in the interface is called **exporting** that function.

The relationship between the client and implementor is illustrated in the following diagram:

client	Interface	Implementation
Responsible for: ● how a function is used	Both sides agree on: ● the function prototype	Responsible for: ● how a function work

Interfaces and header files

In computer science, an interface is a conceptual entity. It consists of an understanding between the programmer who implements a library and the programmer standing between the programmer who implements a library and the programmer who uses it, spelling out the information that is required by both sides. When you write a C program, however, you must have some way to represent the conceptual interface as part of the actual program. In C, an interface is traditionally represented by a header file. You have worked with header files ever since Chapter 2 and have encountered several different ones, including `stdio.h`, `math.h`, and `genlb.h`. Each of these header files specifies the interface to the underlying library.

The distinction between the abstract concept of an interface and the actual header file that represents it may seem subtle at first. In many ways, the distinction is the same as that between an algorithm and a program that implements it. The algorithm is an abstract strategy; the program is the concrete realization of that algorithm. Similarly, C uses header files to provide a concrete realization of an interface.

The same distinction between a general concept and its programming manifestation also comes up in the definition of two other terms that are often used in discussions of interfaces. In computer science, you will often hear the term *package* used to describe the software that defines a library. If you were assigned to develop a library, part of your job would consist of producing a `.h` file to serve as the library interface and one or more `.c` files that together provide an implementation. Those files constitute the **package**. To get a full understanding of a library, however, you must look beyond the software. Libraries embody a specific conceptual approach that transcends the package itself. The conceptual basis of a library is called an **abstraction**.

The relationship between an abstraction and a package is best illustrated by an example. When you write your programs, you use the `printf` function in the `stdio.h` interface for all output operations. For input, you use functions like `GetInteger`, `GetReal`, and `GetLine`, which are made available through the `simpio.h` interface. The `stdio.h` interface provides functions for accepting user input, but they turn out to be more difficult for beginning programmers to use. The two libraries embody different approaches to input operation: the `stdio.h` interface emphasizes power and flexibility while the `simpio.h` interface emphasizes simplicity of structure and ease of use. The approach used in each of these interfaces is part of the abstraction. The associated packages implement those abstractions and make them real, in the sense that they can then be used by programmers.

The contents of header files are discussed in detail in Chapter 8. The best way to get a general impression of how header files are used to represent an interface, however, is to look at an example. The following section walks you through the `graphics.h` header file, which specifies the interface to a simple abstraction for drawing pictures on the screen.

1-2 An introduction to the graphics library

Compared with computer games or commercial word-processing systems, the programs you have worked with so far seem relatively tame. Until now, all programs in this text have displayed their output on the computer screen as numbers and strings formatted by the `printf` function. Modern computer programs typically use the screen in a much more creative way that involves pictures and fancy graphical displays—features that make using the computer both easier and more exciting.

Although graphical displays may make life easier for the user, incorporating them into a program usually makes life more difficult for the programmer. Considered in its entirety, generating even a simple line drawing on the screen is an enormously complex programming problem—well beyond the scope of this text. Luckily, there is no need to look at the problem in its entirety. If you have access to a graphics library, you can ignore the underlying complexity and concentrate instead on high-level operations that cause lines and other graphical features to appear on the screen. The details are hidden on the implementation side of the interface boundary.

To use the graphics library, you must specify its interface by writing the appropriate `#include` line at the top of your program, which in this case is

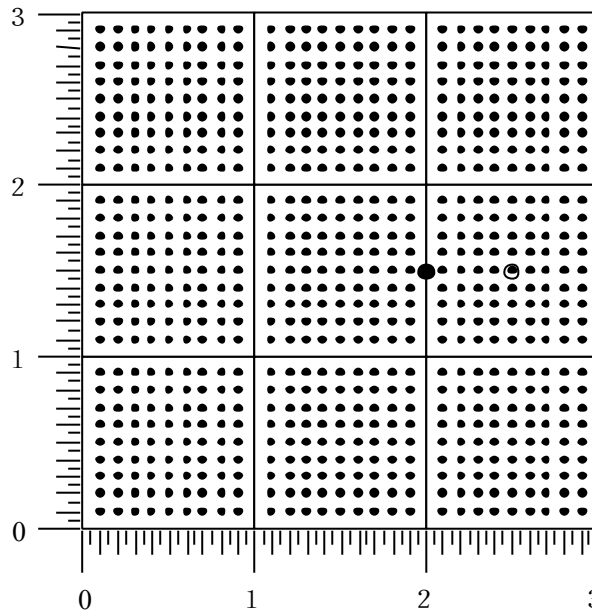
```
#include "graphics.h"
```

The underlying model for graphics.h

Before you can appreciate the procedures and functions available in the `graphics.h` interface, you first need to understand the underlying abstraction. How do you specify positions on the screen? What units should you use for length? These questions are important for understanding the graphical model, which is a central part of the conceptual abstraction.

The graphical capabilities of the display screen you are using depend on the computing hardware you have available. The `graphics.h` interface was designed to be as general as possible, and this generality makes it difficult to describe precisely how the graphical display will be presented on any given system. Typically, when you start up the graphics package, a new rectangular window called the **graphics window** is created on the screen and used as the drawing surface. Whenever you call procedures and functions in the graphics library, the results are displayed in the graphics window.

To specify points within the graphics windows, the graphics library uses an approach that should be familiar from high-school geometry or algebra. All drawing in the graphics window takes place on a conceptual grid as illustrated in Figure 7-1. As in traditional geometry, points are identified by specifying their position relative to the **origin**, which is the point at the lower left corner of the graphics window. The horizontal and vertical lines that emanate from the origin along the edges of the graphics window are called the **axes**; the *x-axis* runs along the bottom of the window and the *y-axis* runs up the left side. Every point within the graphics window is identified by a pair of values, usually written as (x, y) , that specifies the position of that point along the *x* and *y* axes. These values are called the



coordinates of the point. Coordinates are measured in inches relative to the origin, which is the point $(0,0)$. From there, *x* values increase as you move to the right, and *y* values increase as you move up.

FIGURE 7-1 Coordinates in the graphics library

Coordinates in the graphics library come in two forms:

- **Absolute coordinates** specify a point in the window by giving its coordinates absolute coordinates (2.0, 1.5).
- **Relative coordinates** specify a position in the window by indicating how far that point is along each axis from the last position specified. For example, the open dot in Figure 7-1 has absolute coordinates (2.5, 1.5). If, however, you express its coordinates relative to the solid dot, this point is shifted by the relative coordinates (0.5, 0.0). If you wanted to connect these dots with a line, the standard approach would be to specify the first point in absolute coordinates, but then to specify the connecting line in the relative mode.

The best mental model to use for the drawing process is to think of a pen positioned over a piece of transparent graph paper covering the screen. You can move the pen to any location on the screen by specifying the absolute coordinates. Once there, you can draw a straight line by moving the pen relative to its current location with the pen continuously touching the graph paper. From there, you can draw another line beginning from where the last one ended¹.

The functions in the graphics.h interface

The graphics.h interface exports only a small number of functions and procedures for drawing. To draw complicated pictures, you would certainly want more capabilities, but the simplicity of this interface makes it possible to present the entire graphics package in one chapter. You can easily understand it all. The graphics library contains the following functions:

InitGraphics()	Initializes the graphics package
MovePen (x, y)	Moves the pen to an absolute position
DrawLine (dx, dy)	Draws a line using relative coordinates
DrawArc (r, start, sweep)	Draws an arc specified by a radius and two angles
GetWindowWidth ()	Returns the width of the graphics window
GetWindowHeight ()	Returns the height of the graphics window
GetCurrentX ()	Returns the current x-coordinate of the pen
GetCurrentY ()	Returns the current y-coordinate of the pen

These functions provide the capabilities you need to begin drawing simple pictures in the graphics window. To understand how to use them, however, you need to read the documentation for each, which is provided by the interface.

¹ My friends who are artists tell me that they tend to think of a figure like



as a single line, because it is unbroken. In this text, we will always use the term line to mean a segment of a straight line. Thus, the figure in the preceding diagram is actually composed of three lines connected end to end.

The interface for the graphics library is contained in the header file `graphics.h`, shown in Figure 7-2. The header file runs on for several pages, and it is important that you not try to understand it thoroughly all at once. You can read through the initial comments and peruse the file to get a sense of its structure, but it is usually best to use the interface mainly as a reference guide. As new functions are introduced in the text, you should look up the corresponding entries in the interface to see if they make sense.

FIGURE 7-2 `graphics.h` interface

```

/*
 * File: graphics.h
 * -----
 * This interface provides access to a simple library of
 * functions that make it possible to draw lines and arcs
 * on the screen. This interface presents a portable
 * abstraction that can be used with a variety of window
 * system implemented on different hardware platforms.
 */

#ifndef _graphics_h
#define graphics_h

/*
 * Overview
 * -----
 * This library provides several functions for drawing lines
 * and circular arcs in a region of the screen that is
 * defined as the "graphics window." Once drawn, these
 * line and arcs stay in their position, which means that
 * the package can only be used for static pictures and not
 * for animation
 *
 * individual points within the window are specified by
 * giving their x and y coordinates. These coordinates are
 * real numbers measured in inches, with the origin in the
 * lower left corner, as it is in traditional mathematics.
 *
 * The calls available in the package are listed below. More
 * complete description are included with each function
 * description.
 *
 * InitGraphics ();
 * MovePen (x, y);
 * DrawLine (dx, dy);
 * DrawArc (r, start, sweep);
 * width = GetWindowWidth ();
 * height = GetWindowHeight ();
 * x = GetCurrentX ();
 * y = GetCurrentY ();
 */

/*
 * Function: InitGraphics
 * Usage: InitGraphics ();
 * -----
 * This procedure creates the graphics window on the screen.
 * The call to IniGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output. In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics (void);

```

```

/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * -----
 * This procedure moves the current point to the position
 * (x, y), without drawing a line. The model is that of
 * the pen being lifted off the graphics window surface
 * and then moved to its new position.
 */

void MovePen(x, y);

/*
 * Function: DrawLine
 * Usage: DrawLine (dx, dy);
 * -----
 * This procedure draws a line extending from the current
 * point by moving the pen dx inches in the x direction
 * and dy inches in the y direction. The final position
 * becomes the new current point.
 */

void DrawLine (double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc (r, start, sweep);
 * -----
 * This procedure draws a circular arc, which always begins
 * at the current point. The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle. This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on. The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees. If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle. If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point. If sweep is negative, the arc is drawn
 * clockwise from the current point. The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
 *
 * Examples:
 * DrawArc (r, 0, 360)      Draws a circle to the left of the current point.
 * DrawArc (r, 90, 180)    Draws the left half of a semicircle starting from the 12 o'clock position.
 * DrawArc (r, 0, 90)      Draws a quarter circle from the 3 o'clock to the 12 o'clock position.
 * DrawArc (r, 0, -90)     Draws a quarter circle from the 3 o'clock to the 6 o'clock position.
 * DrawArc (r, -90, -90)   Draws a quarter circle from the 6 o'clock to the 9 o'clock position.
 */

void DrawArc (double r, double start, double sweep);

/*
 * Function: GetWindowWidth, GetWindowHeight
 * Usage: width = GetWindowWidth ();
 *         height = GetWindowHeight ();
 * -----
 * These functions return the width and height of the graphics
 * window, in inches.
 */

```



```

double GetWindowWidth (void);
double GetWindowHeight (void);

/*
 * Function: GetCurrentX, GetCurrentY
 * Usage: x = GetCurrentX ();
 *        y = GetCurrentY ();
 * -----
 * These functions return the current x and y positions.
 */

double GetCurrentX (void);
double GetCurrentY (void);

#endif

```

The graphics.h interface contains a few stylized lines that are part of every interface. After the initial comments are the lines

```

#ifndef _graphics_h
#define _graphics_h

```

The very last line in the interface is

```

#endif

```

The purpose of these lines is discussed in Chapter 8, which explains how to write an interface. For the moment, however, you can ignore these lines. Although they turn out to be important to the compiler, they are not at all important to your understanding of how the interface works.

The remainder of the interface consists only of comments and function prototypes. Of these, the comments account for more than 90 percent of the header file. Even though the compiler ignores the comments, they are in many ways the most important part of the interface. The real audience for the interface is not the compiler but the programmer who is trying to write client code. The purpose of the comments is to help programmers understand the abstraction as a whole and use the facilities provided by the interface.

Initializing the package

The first procedure in the graphics.h interface is InitGraphics. As the comments in the interface indicate, this procedure initializes the graphics library and must be called before any other function in the package and before printf is used to display any output on the screen. It is common for a library package to require some initialization. When you use an interface, it is a good policy to red through it to see if it requires any initializing operations.

Drawing straight lines

The function MovePen and DrawLine are the principal line-drawing tools the graphics library offers. As a first illustration, let's draw a single straight line that extends one inch upward from the point (0.5, 0.5). The first step in any main program that uses the graphics

COMMON PITFALLS

Make sure the first line in any program that uses the graphics library is a call to the function InitGraphics. As a more general rule, you should remember that libraries of ten need initialization of some sort. You should therefore check each interface to see whether any initialization is required.

library is always

```
InitGraphics ();
```

To draw the line, you start by moving the pen to the point (0.5, 0.5):

```
MovePen (0.5, 1.0);
```

From there, all you have to do is draw a line in which the x -coordinate does not change at all and the y -coordinate moves one inch up the screen:

```
DrawLine (0.0, 1.0);
```

The complete program is shown in Figure 7-3

FIGURE 7-3 oneline.c

```
/*
 * File: oneline.c
 * -----
 * This program draws a single straight line.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

main()
{
    InitGraphics ();
    MovePen (0.5, 0.5);
    DrawLine (0.0, 1.0);
}
```

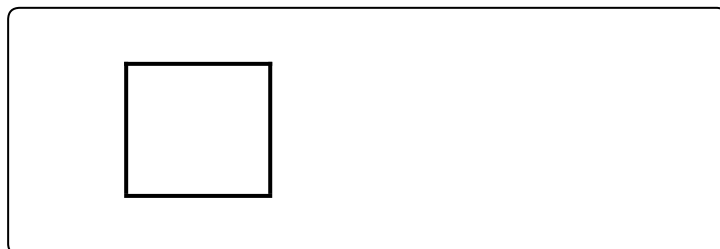
Running the program draws the following picture in the graphics window:



If you want to draw a square instead of a straight line, you can simply add three more calls to DrawLine to the program, so that the main program looks like this:

```
main()
{
    InitGraphics ();
    MovePen (0.5, 0.5);
    DrawLine (0.0, 1.0);
    DrawLine (1.0, 0.0);
    DrawLine (0.0, -1.0);
    DrawLine (-1.0, 0.0);
}
```

which results in picture:



Note that each line begins where the last line ended. This behavior is consistent with the conceptual abstraction of a pen moving around on the surface of the graphics window.

Drawing circles and arcs

The only other drawing function the graphics library provides is `DrawArc`, which you use to create an arc consisting of some fraction of a circle. The proto type for `DrawArc` is

```
void DrawArc(double r, double start, double sweep);
```

Unlike many of the procedure prototypes you have encountered so far, however, the prototype alone is not sufficient for you to understand exactly what this function does. For a complete understanding, you need to look at the comments in the interface as well, which appear in Figure 7-4.

FIGURE 7-4 Interface entry for `DrawArc`

```
/*
 * Function: DrawArc
 * Usage: DrawArc (r, start, sweep);
 * -----
 * This procedure draws a circular arc, which always begins
 * at the current point. The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle. This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on. The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees. If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle. If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point. If sweep is negative, the arc is drawn
 * clockwise from the current point. The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
 *
 * Examples:
 * DrawArc (r, 0, 360)   Draws a circle to the left of the current point.
 * DrawArc (r, 90, 180)  Draws the left half of a semicircle starting from the 12 o'clock position.
 * DrawArc (r, 0, 90)    Draws a quarter circle from the 3 o'clock to the 12 o'clock position.
 * DrawArc (r, 0, -90)   Draws a quarter circle from the 3 o'clock to the 6 o'clock position.
 * DrawArc (r, -90, -90) Draws a quarter circle from the 6 o'clock to the 9 o'clock position.
 */

void DrawArc(double r, double start, double sweep);
```

The first sentence in the comments for `DrawArc` reveals an important piece of information: the arc begins at the current position of the pen. This fact means you have to call `MovePen` to position the pen at the beginning of the arc you wish to draw, just as you do when you begin drawing a line. The comments also give critically important information about what the angles mean and how they are measured. As a client, you need to know this information to use the function successfully. The comments end by offering five examples

that illustrate the use of `DrawArc`. Such examples can be extremely helpful because it is almost always easier to call a function when you have an example to use as a model.

Here, for instance, the documentation suggests that you should be able to draw a complete circle to the left of the current point by using the call

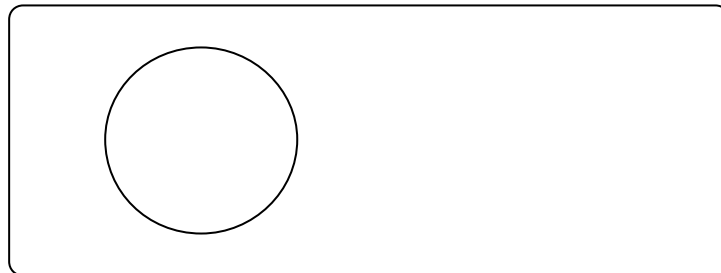
```
DrawArc(r, 0, 360);
```

The arc has radius r and begins at the angle represented by 0 degrees, which is the 3 o'clock position. It extends for 260 degrees, thus creating a complete circle. The starting position on the arc is the current position of the pen when the call is made. Relative to the circle that is drawn, this position is the rightmost point, and the entire circle lies to the left.

Based on this discussion, you should easily be able to write a program that draws a circle with a half-inch radius, centered at the point (1,1). All you have to do is move the pen to the starting point at the right edge of the circle and then call `DrawArc`. The main program is

```
main()
{
    initGraphics ();
    MovePen (1.5, 1.0);
    DrawArc (0.5, 0, 360);
}
```

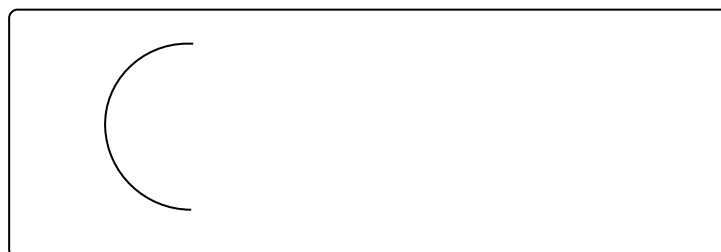
which produces the following display:



To get more of a feeling for how `DrawArc` works, you can try the other examples given in the documentation. For example, replacing the `DrawArc` call in the program by the statement

```
DrawArc(0.5, 90, 180);
```

produces this figure:



Obtaining information about the graphics window

The last four functions exported by the `graphics.h` interface do not actually affect the graphics window but instead return information about it. The functions `GetWindowWidth` and

`GetWindowHeight` return the dimensions of the graphics window, measured in inches. For example, using the statements

```
MovePen (0, 0);  
DrawLine (GetWindowWidth (), GetWindowHeigh ());
```

results in a diagonal line the extends across the entire graphics window.

These functions also enable you can use to center a drawing. The x -coordinate of the center is half of the screen width and the y -coordinate is half of the screen height. You can therefore move the pen to the center of the screen by using the statement

```
MovePen (GetWindowWidth () / 2, GetWindowHeight () / 2);
```

The function `GetCurrentX` and `GetCurrentY` return the x - and y -coordinates of the current pen position. These functions are used primarily in writing higher-level functions and are discussed further in the section on “Switching between absolute and relative coordinates” later in this chapter.

1-3 Building your own tools

The tools introduced in the last section are more useful than you might at first realize. Not only do you have some experience in drawing lines and arcs; you also have begun to put those tools together into program fragments that perform more sophisticated functions. For example, you learned how to put four lines together to make a rectangular box and how to use the `DrawArc` function to make complete circles. But because drawing boxes and circles are common operations, it would be tedious if you had to go through all the steps involved for each box or circle you wanted to draw. It would be more convenient if the designers of the library had simply given you additional tools for drawing boxes and circles. However, whether such functions are explicitly part of the `graphics.h` interface doesn't actually matter. Because C gives you the ability to define your own functions, you can create these tools yourself.

Defining DrawBox

To illustrate this process, suppose you want to define a procedure `DrawBox` that draws a rectangle oriented along the coordinate axes. The first step in the process of writing `DrawBox` is to define its prototype. Doing so is an exercise in design. You know the name of the procedure, but you also have to think about what arguments to include. A useful strategy for figuring out what arguments are required is simply to ask yourself what information the implementation needs. You can't just give it the programming equivalent of the English command: “Draw a rectangle.” The implementation needs to know how big a rectangle and where to put it on the screen. The traditional way for the implementation to get this information is to have the client supply it in the form of arguments.

Even so, there is more than one way to design the `DrawBox` procedure. One possible design for `DrawBox` would be to use only two arguments—width and height—to specify the

dimensions of the box. To indicate the position of the box you would call `MovePen`, after which the box would be drawn relative to that position. Thus, to draw a box at position (x, y) , you would write

```
MovePen(x, y);  
DrawBox (width, height);
```



This example is not the final design.

Another alternative would be to design `DrawBox` to take four arguments— x , y , `width`, and `height`—thereby combining the act of setting the position and that of setting the dimensions, as follows:

```
DrawBox (x, y, width, height);
```

Because the second form is usually more convenient for the caller, it probably makes more sense to adopt the second approach, but either design would certainly work.

In addition to determining how many arguments are required, you must also specify the interpretation of the first two arguments. What does it mean to draw a point (x, y) ? A box does not have an obvious starting point. Where is the point (x, y) relative to the rectangle? One possibility that is convenient for some applications is to implement `DrawBox` so that the point (x, y) specifies the center of the box. A more traditional strategy, however, is to define the origin of the box to be its lower left corner, just as the lower left corner of the graphics window is the origin of the entire coordinate system. The point (x, y) then indicates the position of the origin. No matter how you choose to define the position (x, y) in relation to the box, the main thing you need to do is make sure that the documentation for the function makes your design decision clear.

Thus one possibility for the `DrawBox` prototype is

```
void Drawbox (double x, double y, double width, double height);
```

where x and y specify the origin of the box and `width` and `height` specify its dimensions. Since this procedure is one of your creations and not part of a library, you need to define the implementation as well. The implementation consists simple of the steps necessary to draw the four lines of the figure, expressed in terms of the parameter values. You have already drawn one box in this chapter; all you need to do now is convert the explicit coordinates to the more general, parameter-based form

```
void DrawBox (double x, double y, double width, double height)  
{  
    MovePen (x, y);  
    DrawLine (0, height);  
    DrawLine (width, 0);  
    DrawLine (0, -width);  
    DrawLine (-width, 0);  
}
```

The implementation moves to the origin point for the box and then draws the four line segments necessary to complete the box.

Now that you have this procedure, you can change the implementation of the program to draw a box on the screen so that it uses your new tool. The resulting program, `drawbox.c`, appears in Figure 7-5.

FIGURE 7-5 drawbox.c

```
/*
 * File: drawbox.c
 * -----
 * This program draws a box on the screen.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/* Function prototypes */

void DrawBox (double x, double y, double width, double height);

/* Main program */

main()
{
    InitGraphics ();
    DrawBox (0.5, 0.5, 1.0, 1/0);
}

/*
 * Function: DrawBox
 * Usage: DrawBox (x, y, width, height)
 * -----
 * This function draws a rectangle of the given width and
 * height with its lower left corner at (x, y).
 */

void DrawBox (double x, double y, double width, double height)
{
    MovePen (x, y);
    DrawLine (0, height);
    DrawLine (width, 0);
    DrawLine (0, -height);
    DrawLine (-height, 0);
}
```

The design decision to use the lower left corner as the origin for `DrawBox` does not prevent you from writing other functions that use a different origin. For example, you could also define a function `DrawCenteredBox` whose first two arguments specified the center of the box rather than its corner. If you have already defined `DrawBox`, this new implementation is quite simple to write:

```
void DrawCenteredBox (double x, double y, double width, double height)
{
    DrawBox (x - width / 2, y - height / 2, width, height);
}
```

It is important, however, to be as consistent as you can in your design choices. Using a single model makes it much easier for you, or for anyone else reading your programs, to understand exactly what is going on. In this chapter, functions that draw a figure with respect to some position other than the lower left corner specifically indicate the new origin in the function name, as in `DrawCenteredBox`.

Defining DrawCenteredCircle

It would also be useful to define a new function to draw complete circles. Because circles have no corners, it makes the most sense to define a function `DrawCenteredCircle`, which draws the circle relative to its center. This function needs three arguments: the x - and y -coordinates of the center and the radius r . The prototype for `DrawCenteredCircle` is therefore

```
void DrawCenteredCircle (double x, double y, double r)
{
    MovePen (x + r, y);
    DrawArc (r, 0, 360);
}
```

Although you could use `DrawArc` directly, it is likely that `DrawCenteredCircle` is better tailored to your needs. For one thing, complete circles are quite common in graphical figures and occur more often than partial arcs. For another, using a higher-level function frees you from having to remember exactly how `DrawArc` interprets angles, which you don't have to think about when drawing a complete circle. `DrawCenteredCircle` provides convenience and simplification, both of which are valuable commodities in programming.

Switching between absolute and relative coordinates

The `MovePen` procedure uses absolute coordinates to specify the beginning of a line, which is then drawn by `DrawLine` Using relative coordinates. For some applications, it helps to be able to move the pen to a new position relative to its previous position without drawing a line. Conversely, it is sometimes useful to be able to draw a line to a particular absolute coordinate.

The functions `GetCurrentX` and `GetCurrentY` make it easy to write a relative version of `MovePen` and an absolute version of `DrawLine`. The new functions are called `AdjustPen` and `DrawLineTo`, and their implementations are shown in Figure 7-6.

FIGURE 7-6 Functions `AdjustPen` and `DrawLineTo`

```
/*
 * Function: AdjustPen
 * Usage: AdjustPen (dx, dy);
 * -----
 * This procedure adjusts the current point by moving it
 * dx inches from its current x coordinate and dy inches
 * from its current y coordinate. As with MovePen, no
 * line is actually drawn.
 */

void AdjustPen (double dx, double dy)
{
    MovePen (GetCurrentX () + dx, GetCurrentY () + dy);
}

/*
 * Function: DrawLineTo
```



```

* Usage: DrawLineTo (x, y);
* -----
* This function is like DrawLine, except that it uses the
* absolute coordinates of the endpoint rather than the relative
* displacement from the current point.
*/

void DrawLineTo (double x, double y)
{
    DrawLine (x - GetCurren tX (), y - GetCurren tY ());
}

```

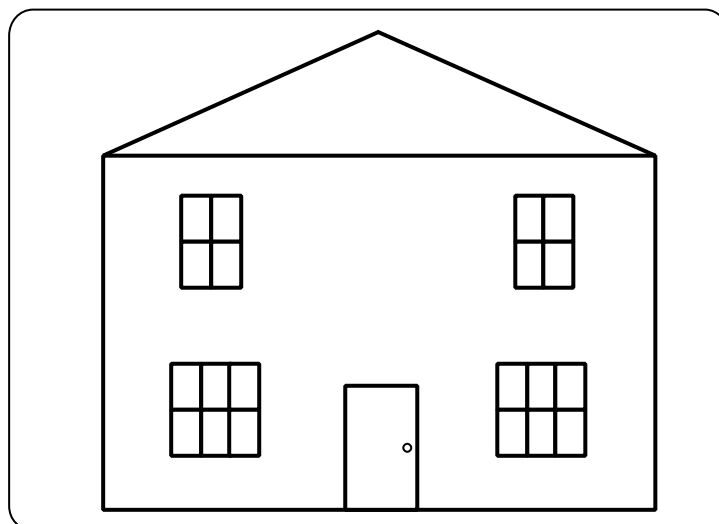
Like `drawBox` and `DrawCenteredCircle`, these functions are not actually part of the graphics library. If you want to use them in your program, you must copy their definitions.

The advantages of defining procedures

As is always the case with procedures, the real advantage of such higher-level tools as `DrawBox` and `DrawCenteredCircle` is not that you can use them a single instance. The big payoff comes from the fact that, once you have defined a new procedure, you can use it over and over again. It is this ability to reuse steps you have written that makes procedures so useful. For example, suppose that you wanted to draw a line of squares across the graphics window, and not just the single square generated by the `drawbox.c` program. You could call `DrawBox` several times in a row, or even put it inside a `for` loop that drew one box each cycle.

1-4 Solving a larger problem

To develop your understanding of the functions in the graphics library, you need to focus your attention on a larger problem. Suppose you have decided to draw a picture of your dream house, using a level of detail that one might find in an elementary-school art class. A house you might draw is shown in the following diagram:



Although the picture has many individual parts, it consists of only two fundamental graphical elements: (1) the straight line, used for the house frame, the door, and the window panes and (2) the circle, used only for the doorknob. If you put these lines and circles together in the right sizes and positions, you can create the complete picture. Moreover, almost all the straight lines are arranged to form boxes, so you can make the most of your new set of tools.

Before you start writing the actual program, however, notice that this specific house has many attributes that define its shape. For example, the house is 3.0 inches wide. The distance from the ground to the attic is 2.0 inches, with another 0.7 inches to the peak of the roof. The door is a rectangle measuring 0.4 inches by 0.7 inches. Each window panes is also a rectangle with dimensions 0.2 by 0.25 inches. Rather than clutter your program with all these numbers, it is useful to give these quantities names, which you can then use in the program. The house diagram shown in the text uses the following constants:

```
#define HouseHeight 2.0
#define HouseWidth 3.0
#define AtticHeight 0.7

#define DoorWidth 0.4
#define DoorHeight 0.7
#define DoorknobRadius 0.04
#define DoorknobInset 0.07

#define PaneHeight 0.25
#define PaneWidth 0.2

#define FirstFloorWindows 0.3
#define SecondFloorWindows 1.25
```

The values are real numbers representing inches, and the names describe their physical meaning in the context of the picture. In the program, the fact that these values are represented using symbolic names makes it easy to change the dimensions if, for example, you wanted a house that was a little wider or had larger windows.

Using stepwise refinement

You are now ready to start the implementation. As discussed in Chapter 5, the best way to approach a large programming problem is to use the strategy of stepwise refinement to break the entire problem down into a set of simpler ones. To apply that strategy to the problem of diagramming the house, you start at the most general level of detail: you want to draw a house. You give that operation a name, such as `DrawHouse`, and define it as a procedure. Implementing the `DrawHouse` procedure becomes your first subproblem. To complete the implementation, you then decompose the entire problem into smaller pieces: drawing the outline, the door, and the windows. Each of these operations then becomes a subproblem at the next level in the decomposition. You carry on this strategy until all the subproblems are reduced to simple operations that fit the tools you have.

As with the `DrawBox` procedure, however, you need to determine whether the `DrawHouse` procedure requires any arguments. The dimensions of the house were specified as

constants in the preceding section. You also need to say where to put the house, so it seems appropriate for the `DrawHouse` procedure to take an x - and a y -coordinate, specifying at what position in the graphics window you want the house to appear. For consistency with `DrawBox`, it makes sense for these values to specify the coordinates of the lower left corner of the house. Thus the prototype for the `DrawHouse` procedure would be

```
void DrawHouse (double x, double y);
```

Calling this procedure instructs the computer to draw a house whose lower left corner is the point (x, y) .

Having defined the prototype, you can now go back and complete the main program. All you really need to do is figure out where the picture of the house should appear on the screen. For example, suppose that you want the house to be centered in the graphics window. As discussed in the section on “Obtaining information about the graphics window” earlier in this chapter, you can use the functions `GetWindowWidth` and `GetWindowHeight` to find the coordinates of the window’s center. For example, if you declare the variables `cx` and `cy`, you can set them to the coordinates of the center by writing

```
cx = GetWindowWidth () / 2;  
cy = GetWindowHeight () / 2;
```

As you have defined `DrawHouse`, however, the diagram it self is drawn relative to the lower left corner and not the center. How can you relate these two positions?

You know that the house is `HouseWidth` inches wide. Thus the left edge of the house must be half that distance from the center. It follows that if you position the left edge of the house at the coordinate

```
cx - HouseWidth / 2
```

the center of the house will end up at the center of the screen. You can repeat the same argument for the y -coordinate. The only difference is that the total height of the house is the sum of the heights of the rectangular structure and the roof. The lower left corner of the house must therefore have the y -coordinate

```
cy - (HouseHeight + AtticHeight) / 2
```

Now that you have the coordinates of the lower left corner of the house, you can finish the implementation of `main` as follows:

```
main()  
{  
    double cx, cy;  
  
    InitGraphics ();  
    cx = GetWindowWidth () / 2;  
    cy = GetWindowHeight () / 2;  
    DrawHouse (cx - HouseWidth / 2, cy - (HouseHeight + AtticHeight) / 2);  
}
```

This definition completes the highest level of the decomposition.

Implementation the `DrawHouse` procedure

At this point, you need to turn your attention to the implementation of `DrawHouse`. Thinking in terms of stepwise refinement, you should already have an idea of what the fundamental operations are. In skeletal form, the `DrawHouse` procedure looks like this:

```
void DrawHouse (double x, double y)
{
    DrawOutline (...);
    DrawDoor (...);
    DrawWindows (...);
}
```

You simply need to fill in the arguments. The procedures `DrawOutline`, `DrawDoor`, `DrawHouse` cannot access the values of the local variables `x` and `y` in `DrawHouse`, so you must pass the coordinate information along to each of the procedures. Choosing exactly what values to pass, however, requires some thought. The outline starts at the same corner as the house, so the `x` and `y` values there are probably the same. For the door, you might want to compute the coordinates of the door itself and then pass these coordinates to `DrawDoor`. Because there are several windows drawn relative to the house frame, the `DrawWindows` function should probably take the house coordinates as arguments, although it will compute more specific coordinates for each of the windows as part of its implementation. If you implement the `DrawHouse` procedure as suggested, it will come out looking like this:

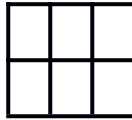
```
void DrawHouse (double x, double y)
{
    DrawOutline (x, y);
    DrawDoor (x + (HouseWidth - DoorWidth) / 2, y);
    DrawWindows (x, y);
}
```

Looking for common patterns

When you approach large problems, stepwise refinement is only one of several strategies you can use to your advantage. Another extremely useful strategy involves trying to find common elements within the different parts of a large problem so that you can apply a single solution technique to all of them. In essence, this approach consists of determining what tools would be best for the job. For example, if you could solve several parts of a problem easily using one procedure that performed a particular operation, it might be worth creating that procedure.

If you think about the problem of framing the house from this perspective, there are several tools you might want, some of which you have already written. For example, the outline of the house is a box, as are the door frame and the windows, so the `DrawBox` tool should prove very handy. The doorknob is a circle, which suggests the use of `DrawCenteredCircle`. But you should also think about other tools that would help. The roof of the house is a triangle. Even though there is only one triangle in the picture, writing a `DrawTriangle` procedure might be worthwhile, particularly since you would then have it available for use in other programs. It is even more important, however, to notice the regular structure of the windows and consider the possibility of writing a more general procedure that can draw each of the different sets of windows.

To design a tool that is appropriate for the windows, it pays to generalize the problem as much as you can. The more general your tools are, the easier it is to apply them to a wide variety of circumstances. One way to ensure that the tools you build are widely applicable is to step back from the specific characteristics of the current problem and try to visualize the necessary operations at a higher, more abstract level. In the context of the house, the picture



depicts a windows with several panes. When you focus on this figure by itself, however, what you see is simple a rectangular grid composed of two rows, each of which contains three boxes. If you had a procedure `DrawGrid` that drew a rectangular grid, you could use that procedure to draw each set of windows.

What arguments does `DrawGrid` require? To achieve the necessary generality, you have to make sure that the `DrawGrid` procedure does not refer to the particular context of the house. Because using the constants `PaneWidth` and `PaneHeight` would make the procedure specific to the house picture, it is better to have the caller pass the width and height of each box within the grid as arguments. The caller knows that it is drawing a window and can supply `PaneWidth` and `PaneHeight` for this specific application. The procedure itself is just drawing boxes. Besides the height and width of each box within the grid, `DrawGrid` also needs to know the coordinate position of the grid as a whole. To be consistent with the other tools, `DrawGrid` should interpret these coordinates as representing the lower left corner of the grid. Finally, the procedure must know the number of columns and rows in the grid. Thus, the prototype for `DrawGrid` should look like this:

```
void DrawGrid (double x, double y, double width, double height, int columns, int rows);
```

Given that you already have the function `DrawBox`, the implementation of `DrawGrid` is reasonably straightforward. The implementation consists of a pair of nested `for` loops that calls `DrawBox` for each column within each row, as follows:

```
/*
 * Function: DrawGrid
 * Usage: DrawGrid (x, y, width, height, columns, rows);
 * -----
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid. As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid (double x, double y, double width, double height, int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox (x + i * width, y + j * height, width, height);
        }
    }
}
```

Given the implementation of `DrawGrid`, you can construct each of the window patterns just by calling `DrawGrid` with the appropriate arguments.

Even though it is usually best to design a problem from the top down, it is often best to *implement* it from the bottom up. Implementing the low-level tools first makes it much easier to debug the individual pieces of your program, which is usually easier than trying to debug all of it at once. This strategy is called **bottom-up implementation**.

Finishing the decomposition

Given the new `DrawGrid` tool, the rest of the program to draw the house is straightforward example of stepwise refinement. The complete program, `house.c` appears in Figure 7-7.

FIGURE 7-7 `house.c`

```
/*
 * File: house.c
 * -----
 * This program draws a simple frame house.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * The following constants control the sizes of the
 * various elements in the display.
 */

#define HouseHeight 2.0
#define HouseWidth 3.0
#define AtticHeight 0.7

#define DoorWidth 0.4
#define DoorHeight 0.7
#define DoorknobRadius 0.04
#define DoorknobInset 0.07

#define PaneHeight 0.25
#define PaneWidth 0.2

#define FirstFloorWindows 0.3
#define SecondFloorWindows 1.25

/* Function prototypes */

void DrawHouse (double x, double y);
void DrawOutline (double x, double y);
void DrawWindows (double x, double y);
void DrawDoor (double x, double y);
void DrawBox (double x, double y, double width, double height);
void DrawCenteredCircle (double x, double y, double r);
void DrawGrid (double x, double y, double width, double height, int columns, int rows);
```

```

/* Main program */

main()
{
    double cx, cy;

    InitGraphics ();
    cx = GetWindowWidth () / 2;
    cy = GetWindowHeight () / 2;
    DrawHouse (cx - HouseWidth / 2, cy - (HouseHeight + AtticHeight) / 2);
}

/*
 * Function: DrawHouse
 * Usage: DrawHouse (x, y);
 * -----
 * This function draws a house diagram with the lower left corner
 * at (x, y). This level of the function merely divides up
 * the work.
 */

void DrawHouse (double x, double y)
{
    DrawOutline (x, y);
    DrawDoor (x + (HouseWidth - DoorWidth) / 2, y);
    DrawWindows (x, y);
}

/*
 * Function: DrawOutline
 * Usage: DrawOutline (x, y);
 * -----
 * This function draws the outline for the house, using (x, y)
 * as the origin. The outline consists of a box with a triangle
 * on top
 */

void DrawOutline (double x, double y)
{
    DrawBox (x, y, HouseWidth, HouseHeight);
    DrawTriangle (x, y + HouseHeight, HouseWidth, AtticHeight);
}

/*
 * Function: DrawDoor
 * Usage: DrawDoor (x, y);
 * -----
 * This function draws a door, with its doorknob. As usual,
 * (x, y) specifies the lower left corner of the door.
 */

void DrawDoor (double x, double y)
{
    DrawBox (x, y, DoorWidth, DoorHeight);
    DrawCenteredCircle (x + DoorWidth - DoorknobInset, y + DoorHeight / 2, DoorknobRadius);
}

/*
 * Function: DrawWindows
 * Usage: DrawWindows (x, y);
 * -----
 * This function draws all the windows for the house,
 * taking advantage of the fact that the windows are all
 * arranged in two-dimensional grids of equal-sized panes.
 * By calling the function DrawGrid, this implementation

```

```

* can create all of the window structures using a single
* tool.
*/

void DrawWindows (double x, double y)
{
    Double xleft, xright;

    xleft = x + HouseWidth * 0.25;
    xright = x + HouseWidth * 0.75;
    DrawGrid (xleft - PaneWidth * 1.5, y + FirstFloorWindows, paneWidth, PaneHeight, 3, 2);
    DrawGrid (xright - PaneWidth * 1.5, y + FirstFloorWindows, PaneWidth, PaneHeight, 3, 2);
    DrawGrid (xleft - PaneWidth, y + SecondFloorWindows, PaneWidth, PaneHeight, 2, 2);
    DrawGrid (xright - PaneWidth, y + SecondFloorWindows, paneWidth, PaneHeight, 2, 2);
}

/*
* Function: DrawBox
* Usage: DrawBox 9x, y, width, height)
* -----
* This function draws a rectangle of the given width and
* height with its lower left corner at (x, y).
*/

void DrawBox (double x, double y, double width, double height)
{
    MovePen (x, y);
    DrawLine (0, height);
    DrawLine (width, 0);
    DrawLine (0, -height);
    DrawLine (-width, 0);
}

/*
* Function: DrawTriangle
* Usage: DrawTriangle (x, y, base, height)
* -----
* This function draws an isosceles triangle (i.e., one with
* two equal sides) with a horizontal base. The coordinate of
* the left endpoint of the base is (x, y), and the triangle
* has the indicated base length and height. If height is
* positive, the triangle points upward. If height is negative,
* the triangle points downward.
*/

void DrawTriangle (double x, double y, double base, double height)
{
    MovePen (x, y);
    DrawLine (base, 0);
    DrawLine (-base / 2, height);
    DrawLine (-base / 2, -height);
}

/*
* Function: DrawCenteredCircle
* Usage: DrawCenteredCircle (x, y, r);
* -----
* This function draws a circle of radius r with its
* center at (x, y).
*/

void DrawCenteredCircle (double x, double y, double r)
{
    MovePen(x = r, y);
    DrawArc (r, 0, 360);
}

```



```

/*
 * Function: DrawGrid
 * Usage: DrawGrid (x, y, width, height, columns, rows);
 * -----
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid. As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid (double x, double y, double width, double height, int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox (x + i * width, y + j * height, width, height);
        }
    }
}

```

SUMMARY

In this chapter, you have started to explore the concept of an interface, which is one of the most powerful ideas in modern programming. An interface is the point of connection between the implementor of a library abstraction and its clients. The interface specifies the information that both sides need to know. You will learn more about interface—along with strategies for designing them—in Chapter 8.

This chapter also presents a particular interface—`graphics.h`—to serve as a general example of how interfaces work. The `graphics.h` interface makes it possible for you to draw simple picture by positioning lines and arcs on the screen. Along with the functions provided by the interface itself, you have also learned how to write additional high-level tools, such as `DrawBox`, `DrawCenteredCircle`, and `DrawGrid`, that extend the power of the `graphics` library.

Important points introduced in this chapter include:

- The code used to represent a library is collectively called a package. The package is the programming manifestation of an abstraction, which is the underlying conceptual basis for the library.
- The functions in a library are written by *implementors* and are called by *clients*. The point at which clients and implementors come together is called the *interface*.
- Interfaces in C are represented using *header files*. Header files used as interfaces contain extensive documentation as well as the prototypes for the functions *exported* by the library.
- The `graphics` library makes it possible for you to draw pictures on the screen. The facilities in the library are described in the `graphics.h` interface, which appears in Figure 7-2.
- Pictures drawn using the `graphics` library consist of lines and arcs that appear in the *graphics window*. Coordinates in the library are specified in inches relative to the origin, which is the lower left corner of the graphics window. The model used for the

library is that of a pen moving across the screen.

- Before using any of the other functions in the graphics library, you must first call `InitGraphics` to initialize the package.
- To draw a line segment, you first call `MovePen` to position the pen at the starting point and then call `DrawLine` to draw the actual line. The arguments to `MovePen` are specified as *absolute coordinates*, indicating a particular position in the graphics window. The arguments to `DrawLine` are *relative coordinates*, which indicate how far the pen moves from its previous position. Once you have drawn one line, you then can draw additional lines, each of which begins where the last one ended. To draw a line in a new position, you must again call `MovePen` to position the pen at the new starting point.
- To draw a circular arc, you first call `Move pen` to position the pen at a point on the circle and then call `DrawArc`. The arguments to `DrawArc` are the radius of the circle, the angle at which the arc begins, and the number of degrees in the arc.
- You can extend the capabilities of the library by defining new functions, such as `DrawBOx` and `DrawCenteredCircle`.
- Stepwise refinement is a critically important tool for solving large graphics programs, just as it is for other types of programming.
- Another useful strategy for working with large programs is to think about general tools that would be applicable to the current problem. If you then build those tools, you can more easily solve the current problem as well as others that involve similar operations. When you write the program itself, it is usually best to build these tools first so that you can test your program in pieces as you go. This approach is called *bottom-up implementation*.

REVIEW QUESTIONS

1. True or false: Everything you need to know about interfaces has been covered in this chapter.
2. Define the following terms: *interface*, *package*, *abstraction*, *implementor*, *client*.
3. What is the difference in perspective between the implementor and the client?
4. How are interfaces represented in C?
5. What goes into a C header file?
6. Why are comments particularly important in header files?
7. How are coordinates measured in the graphics library? What is meant by the term origin, and where is it in the graphics window?
8. Describe the difference between absolute and relative coordinates.
9. What are the eight functions exported by the graphics library?
10. When you use the graphics library, what statement should appear at the beginning of the main program?
11. What function in the graphics library do you use to change the position of the pen?
12. What statements would you write to draw a line from the origin to the point (2,1)?
13. What does it mean if the third argument to `DrawArc` is negative?
14. Describe the arcs produced by each of the following calls to `DrawArc`:

- a. `DrawArc (1.0, 0, 270);`
 - b. `DrawArc (1.0, 135, -90);`
 - c. `DrawArc (1.0, 180, -45);`
 - d. `DrawArc (1.0, -90, 180);`
15. On a piece of graph paper, sketch an approximation of the shape that would be produced by the following statements:
- ```

MovePen (1.0, 1.0);
DrawArc (4.0, -15, 2 * 15);
DrawArc (4.0, 180 - 15, 2 * 15);

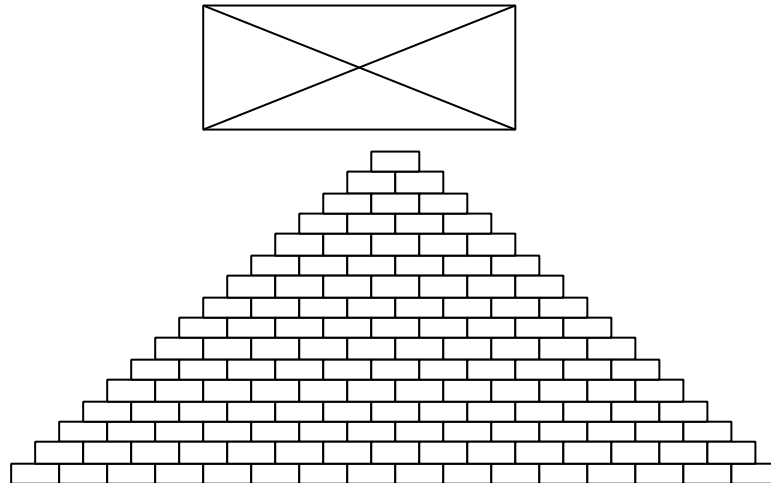
```
16. How do you obtain the coordinates of the center of the graphics window?
17. What are the advantages of implementing new procedures like `DrawBox` and `DrawCenteredCircle`?
18. When you design a function for use as a tool, why is it useful to step outside the specific problem domain and consider the problem more abstractly?
19. What is meant by bottom-up implementation? What are its advantages?

## PROGRAMMING EXERCISES

1. Write a function `DrawCrossedBox` that takes the same arguments as `DrawBox` but also draws lines along the diagonals of the rectangle. For example, the call

```
DrawCrossedBox (0.5, 0.5, 1.0, 0.5);
```

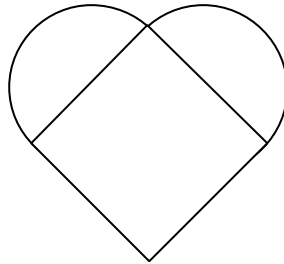
should produce the figure with the lower left corner of the rectangle at the point (0.5, 0.5).



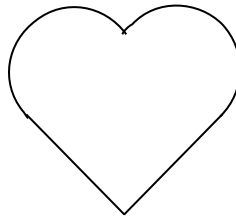
2. Write a program that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as shown in the following diagram:

Your implementation should use the constant `NBricksInBase` to specify the number of bricks in the bottom row and the constants `BrickWidth` and `BrickHeight` to specify the dimensions of each brick.

One way to draw a heart-shaped figure is by drawing two semicircles on top of a square that is positioned so that its sides run diagonally, all illustrated by the following diagram:

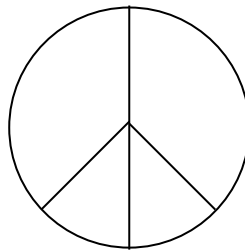


Write a program that uses this construction to draw a heart on the screen. Your program should display the heart without drawing the interior lines that form the top of the square, so the output looks like this:



4. In the 1960s, this symbol

Became universally identified as the *peace symbol*, and it still shows up from time to time as a motif for T-shirts or jewelry. The peace symbol took its form from the letters N and D—the initial letters in nuclear disarmament—as expressed in the international



semaphore code:



The peace symbol is formed by superimposing the lines in these two diagrams (without the flags) and enclosing them in a circle.

Implement a function `DrawPeaceSymbol` with the prototype

```
void DrawPeaceSymbol (double x, double y, double r);
```

that draws a peace symbol centered at the point  $(x, y)$  with a circle of radius  $r$ . Write a main program to test your function.

5. The sample runs in this text are represented by enclosing the output from the computer inside a box with rounded corners. Implement a function `DrawRoundedBox` to draw such boxes using the graphics library. The function should take exactly the same arguments as `DrawBox` but should replace the corners with quarter circles of a constant radius given by

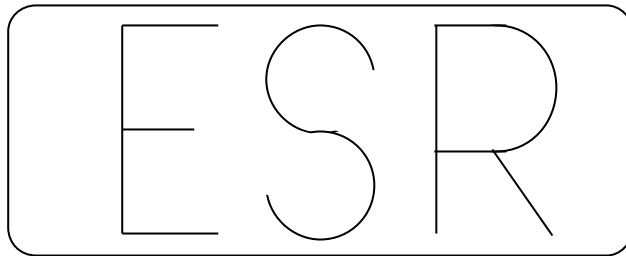
```
#define CornerRadius 0.2
```

for example, calling `DrawRoundedBox` with a width of 1.0 inch and a height of 0.6 inch should produce this figure:



Make certain that your function behaves in a reasonable way if the height or width is less than `CornerRadius`.

6. Write a program to draw your initials on the graphics window. For example, if I wrote this program, I would want the output to be



You'll need to think about the best decomposition to use in writing the program. Imagine that you've been asked to design a more general letter-drawing library. How would you want the functions in that library to behave in order to make using them as simple as possible for your clients?

7. Write a program that draws a picture of the Halloween pumpkin shown in the following

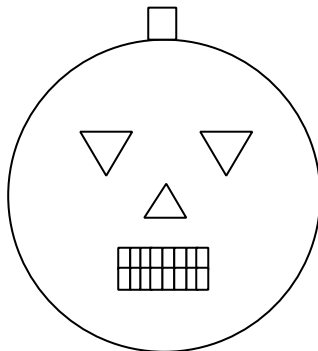


diagram:

As in the `house.c` program shown in Figure 7-7, your picture should be controlled by

several constants:

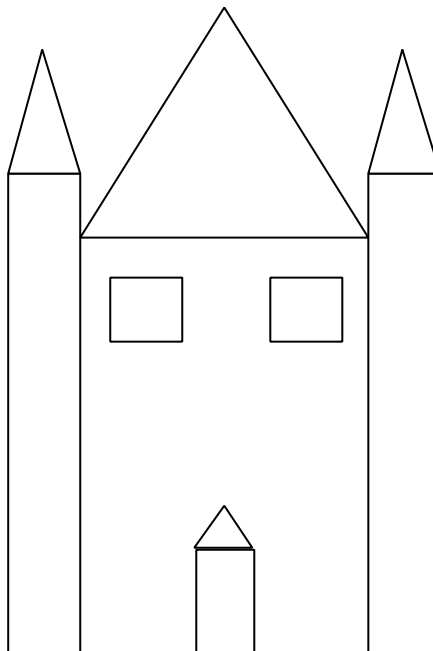
```
#define HeadRadius 1.0
#define StemWidth 0.1
#define stemHeight 0.15
#define EyeWidth 0.3
#define EyeHeight 0.2
#define Noseidth 0.2
#define NoseHeight 0.2
#define NTeethPerRow 7
#define ToothWidth 0.083333
#define ToothHeight 0.15
```

These values are the ones used to produce the pumpkin shown in the diagram, and you should be able to figure out what each constant means by looking at the picture. Your program must be written so that changing any of these constants changes the picture in the appropriate way. For example, if you change `NTeethPerRow` to 4, the new diagram should have only four teeth in each row, but the mouth should still be centered horizontally. The two eyes and the mouth of the pumpkin face should be drawn halfway from the center to the edge of the circle in the appropriate direction, so that changing `HeadRadius` also changes the positions at which these features are drawn. The center of the circle representing the pumpkin should appear at the center of the screen.

8. If you wanted a house to go along with the Halloween pumpkin you designed in exercise 7, you might want to draw a diagram of the House of Usher, which Edgar Allen Poe describes as follows:

I looked upon the scene before me...upon the bleak walls—upon the vacant eye-like windows...with an utter desperation of soul...

From Poe's description, you might image a house that looks something like this:



Write a program that draws the house illustrated in the diagram, using the following constants to specify the various dimensions:

```
#define HouseWidth 1.5
#define HouseHeight 2.0
```

```

#define HouseArch 1.0

#define TowerWidth 0.4
#define TowerHeight 2.3
#define TowerArch 0.6

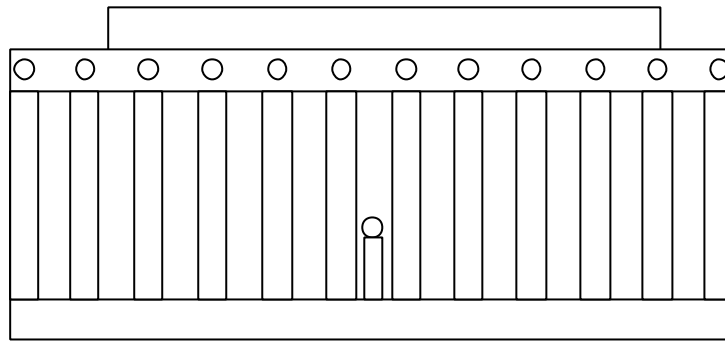
#define DoorWidth 0.3
#define DoorHeight 0.5
#define DoorArch 0.25

#define WindowLevel 1.4
#define WindowSize 0.3

```

The constants whose names end in Arch specify the height of the triangular portion on top of the rectangular base, the windows are assumed to be square and therefore have only the single dimension WindowSize.

9. Write a program that draws the following stylized picture of the Lincoln Memorial in Washington, D.C.:



As in exercises 7 and 8, your program should use the following constants to define the characteristics of the picture:

```

#define MemorialWidth 4.0
#define Pedestalheight 0.3

#define NumberOfColumns 12
#define ColumnWidth 0.14
#define ColumnHeight 1.0
#define ColumnCircleRadius 0.05

#define LowerRoofHeight 0.3
#define UpperRoofWidth 3.5
#define UpperRoofHeight 0.3

#define StatueEidth 0.1
#define StatueHeight 0.2

```

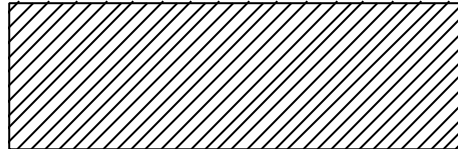
Note that NumberOfColumns is one of the constants that define the picture. In designing your program, you should be sure that it is possible to change the value of NumberOfColumns and still have the columns come out equally spaced across the width of the memorial.

10. Write a function DrawShadedBox that draws a box whose interior is shaded by closely spaced diagonal lines throughout the entire figure. The function should take a fifth

parameter (after the four used in DrawBox) that specifies the distance between each of the shading line, measured along the edges of the box rather than diagonally,. The shading separation parameter should be an integer measured in **points**, a unit of measure favored by printers and typesetters that is equal to 1/72 of an inch. For example, the function call

```
DrawShadeBox (1.0, 1.0, 2.0, 0.75, 5);
```

should produce the following output:



11. Use the DrawShadedBox function from exercise 10 as part of a main program to draw a checkerboard, for which the edge length of each square is 0.25 inches and the sanding separation for the dark squares is 3 points. The program should result in the following figure:

