# Chapter 8

# Libraries

A library is a set of functions, packaged as a system resource, and intended for use by other programs on the system. Normally a library is not written for one single program, it is intended to be used by many programs. By packaging the functions as a system resource, the code does not need to be rewritten for every program that uses it. Although a library may include any number of functions, it does not include one important function: main(). A library is not an executable program. A library consists of functions that can be combined with a main() to form a complete executable program.

On a typical computing system, there are many libraries. A single library tends to contain functions concerning a single topic, such as mathematical calculation, memory debugging, network operations, or graphics. Some libraries are small, containing ten or so functions, while other libraries may contain a thousand functions. Some libraries are considered standard, having become common to a large number of computing systems. Examples include the C standard library and the X library. Some libraries are developed by individuals or companies to support their specific product line, and may only be found on computing systems related to those products.

When using a library, a program does not need to use every function inside it. A program may call only a single function within that library, or it may use them all, or any number in between. A program may use multiple libraries. A library may be built on top of another library, calling upon its functions. In this case, a program using the top level library must also make use of the lower level library. Graphics libraries, in particular, have developed this way.

This chapter covers libraries from the perspective of a system resource.

A serious programmer must know how and when to use libraries. Using a library saves time in programming, because a programmer can make use of existing code. Library code tends to be written by experts, so that it tends to have good design and performance. Because a library is used by many programmers, it is usually debugged by a wide audience, and so a programmer can use it with confidence.

Basic knowledge of some of the common libraries is also useful. In order to understand how libraries work, and to become comfortable with them, this chapter will describe three libraries in some detail. However, the coverage of these libraries is not intended to turn the reader into an expert with those particular libraries. Rather, they are intended to familiarize the reader with the process of using a library. A programmer typically only becomes an expert with a specific library through extensive code or product development that makes use of that library. That is generally a goal only when tackling a specific job.

## 8.1 Using a library

There are two basic steps to using a library. First, one or more header files must be included in the C program code. Second, the library must be linked into the executable. These concepts can be demonstrated with the following code example:

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

main()
{
double x,s;
s=8.0;
x=sqrt(s);
printf("%lf\n",x);
}
```

In this example we are making use of the sqrt() function, which is one of the functions in the math library. Assume that this code is stored in a file named `sq.c`. First, the header file `math.h` is included in order to use the math library. Second, when compiling, we must link to the math library:

```
gcc -o sq sq.c -lm
```

The command line argument `-lm` tells the compiler to link (`-l`) to a library file named `m`. This library file is what actually contains the code for the sqrt() function, and all the other functions in the math library. In the following two sections we take a look at what is inside a header file, and why it is needed; and at a library file, and how it works.

### 8.1.1 Header files

A header file does not contain the code for any of the functions in the library. That code is contained in the library file, which is brought in during linking. Why then do we need to include the header file? For example:

```
#include <math.h>
```

One can think of a header file as the *instructions for how to use the library.* It contains function prototypes, which describe the inputs and outputs of all the functions (how many and what types of parameters each function takes, and what type of value each function returns). For example, in math.h we can trace the following code[1]:

```
double sqrt(double x);
```

This prototype tells us that the the sqrt() function takes in one argument, a double, and also returns a double. By including the header file into our own program, we inform the compiler of how the function works, so that it can properly compile our use of the function. Remember, our program does not include the code for the sqrt() function. Therefore the compiler needs the function prototype in order to properly align our code which calls the function.

A header file can also contain constants. For example, within math.h we can find the following code:

```
#define M_PI           3.14159265358979323846  /* pi */
```

[1]Function prototypes are often written using nested preprocessing substitutions, to provide for flexibility in implementation and system independence. However, the net effect of expanding the preprocessing substitutions is to produce a line of code like the one given here.

This provides a constant value for pi, often used in trigonometric and other mathematical calculations. A programmer can use this constant without having to redefine it for every program. Within the header file X.h, the primary include file for the X library, we find another use for constants:

```
#define KeyPress                2
#define KeyRelease              3
#define ButtonPress             4
#define ButtonRelease           5
#define MotionNotify            6
/* ... list continues for 34 entries ... */
```

These constants provide plain-English phrases for values commonly passed to and from functions within the library. This particular list continues, defining 34 different possible values for a common function parameter. Programmers typically find it easier to remember text phrases, as opposed to numeric values, for oft-used parameter values. For example, one could write the following code:

```
if (SomeEvent.type == 2)
  /* process key press event */
```

However, it is more common to write that code as follows:

```
if (SomeEvent.type == KeyPress)
  /* process key press event */
```

This code takes advantage of the constant definitions in the header file to make the code easier to write, and more readable.

A header file may use typedef and struct definitions to create library-specific aliases for common data types, or new data types. For example, within the X.h header file, we find:

```
typedef unsigned long Mask;
```

This code creates an alias called "Mask" for the unsigned long int. Why is this done? The X library uses bitwise operations to send and receive data through many functions. Since a bitmask will be used as a parameter for many of these functions, the X library provides a data type named "Mask"

to promote code readability, by more strongly identifying what a particular variable is intended to do.

Another example can be seen in the `FILE` data type. By including the header file stdio.h, we eventually find the following lines of code:

```
struct _IO_FILE {
  int _flags;
  int _fileno;
  int _blksize;
  /* ... many additional fields not printed here ... */
  }

typedef struct _IO_FILE FILE;
```

This code defines a structure that contains information about accessing a file. The code then defines an alias for that structure, to simplify writing code. These lines of code explain the commonly seen:

```
#include <stdio.h>

FILE *fpt;
```

First, without including the stdio.h header file, the compiler will not understand the keyword "FILE". Second, by tracing through the definition we find that the variable `fpt` is nothing more than a pointer to a structure. When using a library, it is common to make use of seemingly exotic and unknown data types. However, they are nothing more than typedefs, aliases and structure definitions, written out within the header file, to make code more readable and portable.

Header files for the C standard libraries are usually stored in /usr/include on a linux system (or any unix-based system). On an MS Windows system, it depends on which compiler is being used. Different compilers store the header files in different locations. The MS Visual C compiler typically places those header files in `C:\Program Files\Microsoft Visual Studio\VC98 \Include`. It does not particularly matter where header files are placed, so long as the compiler knows where to find them. By default a compiler will look in its preferred location(s), defined during installation. If a header file is placed in a different location, for example by installing a new library in

a non-standard location, then the compiler must be told where to find the header file. Using the gcc compiler, this is accomplished by using the `-Ipath` command line argument. For example:

```
gcc -o sq sq.c -I/usr/include/mathlib -lm
```

The option `-I/usr/include/mathlib` tells the compiler to look in the /usr /include/mathlib directory, in addition to the standard locations, for any requested include files. We will see this again when we look at the X library.

### 8.1.2 Library files

A library file contains the actual code for the functions in the library. During compiling, we must link to the library file to bring the code together with our own, to make the executable program. In the example at the beginning of this section, we used the command line argument `-lm` while compiling to tell the compiler to link to the `m` file, which is the math library file. But where is this `m` file, and what exactly is inside it?

On a linux system (and all unix-based systems), library files are typically stored in /usr/lib. Linux systems use the following convention for naming library files: they begin with the letters `lib`, and have a filename suffix of `.a`. The only part of a library filename that is unique lies in between those parts. Thus, the math library file, which we called `m` when compiling, is actually named `libm.a` on the system. We can find it as follows:

```
ls -l /usr/lib/libm.a
-rw-r--r--  1 root  root  3092430  Sep 4 2001 /usr/lib/libm.a
```

Notice that the file is fairly large, about 3 MB (this will vary from system to system). This shows that there is quite a bit of code in the library file. The math library contains dozens of functions, some of which are quite complex.

On an MS Windows system, library files have no fixed prefix but they do all end with either the `.lib` or `.dll` suffix. They may be found in several directories, including `C:\Winnt\system32`, `C:\Winnt\system`, and a `\lib` subdirectory installed as part of a compiler (for the MS Visual C compiler, this would be `C:\Program Files\Microsoft Visual Studio\VC98\Lib`).

When linking, a compiler knows to look for library files in the standard directories, usually defined when the compiler is installed. It is also aware

of any naming conventions, such as expanding `m` to `libm.a`. However, some library files may be stored in non-standard directories. For example, a new library may be added to a system and stored in its own folder, in order to make maintenance of the library easier. An example of this is the X library. It is commonly stored in /usr/X11R6, with subdirectories for its include (/usr/X11R6/include) and library (/usr/X11R6/lib) files. In order to link with that library, we must tell the compiler to look in that directory, in addition to the standard directories, when looking for library files:

```
gcc -o xprog1 xprog1.c -lX11 -L/usr/X11R6/lib
```

The command line argument `-L/usr/X11R6/lib` tells the compiler to add the path /usr/X11R6/lib to the set of directories in which to find library files.

A library file contains the actual code for all the functions in the library. The code for the library functions is static, in the sense that it is not expected to change (ignoring for the moment library upgrades). Therefore, it is pre-compiled and stored in an intermediate format called a library file. For the present discussion, the detailed format of a library file is not important; it is enough to know that it is code that has previously been compiled and is ready to be linked. If one tries to open the file /usr/lib/libm.a with a text editor, it will look like garbage, since it is not source code (ASCII text). This topic will be visited again in the chapter on program building, when we take a deeper look at object code files.

## 8.2 Purpose of libraries

There are several reasons to package a set of functions into a library:

**Convenience, repetition.**

An example in this category is the string function library. Many of the string functions are easy to code. For example, the strlen() function is only a couple lines of code. However, string functions are used frequently, and even though they may be easy to code, it is convenient to put them in a library to avoid rewriting them every time a new program is written.

**Difficult to code.**

An example in this category is the math library. The functions in the

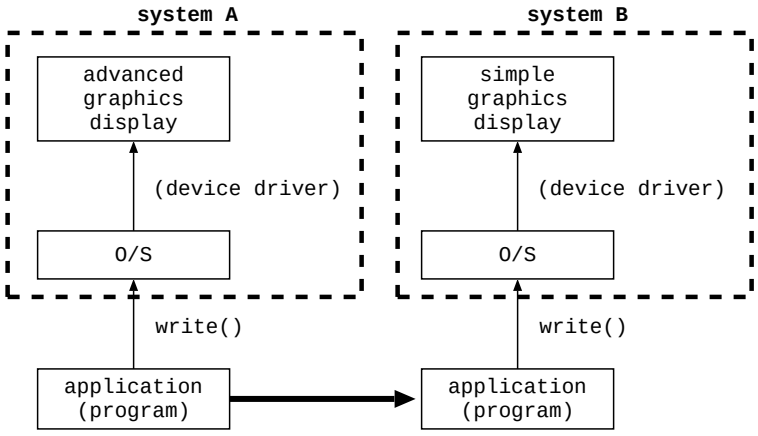


Figure 8.1: A program must be written to work with multiple device drivers if it is intended to work on different hardware or systems.

math library, such as sqrt() and cos(), are iterative in nature and very difficult to code. For example, to solve for the square root of a number, one could continually multiple a number by itself, lowering or raising the value, until it is close enough to the value whose square root is being sought. Because these functions are difficult to write, we prefer to utilize the expertise of people who have studied these problems extensively, and written code for us to use. While we might be able to write a method that works, the experts have written more efficient, precise methods based on a detailed study of computational mathematics.

**Hardware/system independence.**

An example in this category is a graphics library, such as the X library or the OpenGL library. In order to access a piece of hardware, a program must go through a device driver in the O/S (see earlier chapter on I/O). The program can call the open function for the specific piece of hardware, and then call the write function to send it data. If we were only developing an application for one system (defined as the O/S plus hardware), then this is a viable method for graphical output. However, most of the time we want an application to be capable of running on a variety of graphics displays or graphics cards.

Figure 8.1 shows an example. Suppose that on system A we have a state-of-the-art graphics card, that can render 3D primitives with shaded lighting,

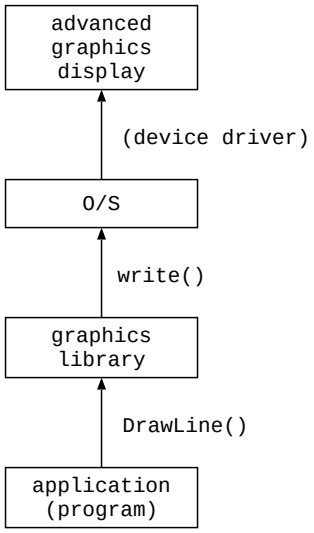


Figure 8.2: A graphics library allows an application to call system or hardware-independent functions.

textures, and other advanced features. System B on the other hand has an inexpensive, low-resolution, straight pixel display card. Each of these systems uses a different device driver, specific to its hardware. The data that is sent via a write() function call on system A will look very different from the data that is sent using write() on system B. Should the application need to know about different device drivers, and change how it calls write() depending on what hardware is available?

Instead, we use a graphics library to perform this job. Figure 8.2 demonstrates the process. The graphics library contains generic graphics functions, such as "DrawLine()". Within its functions, a graphics library implements the code specific to different graphics hardware to carry out that operation. The details of how and when the graphics library calls write() to actually implement DrawLine() are hidden from us. This is very similar to how the details of the write() function call are hidden in the device driver.

Graphics libraries primarily provide us with hardware independence, but they can also provide us with O/S independence. Some of the more generally accepted and popular graphics libraries are available on a variety of operating systems, and support a large variety of hardware. Examples include the X library and the OpenGL library.

| header file | contents |
|---|---|
| stdio.h | I/O functions, such as printf() and scanf() |
| stdlib.h | large variety of functions, including memory allocation |
| string.h | the string functions, such as strlen() and strcpy() |
| math.h | the math functions, such as fabs() and sqrt() |
| time.h | functions for converting various time and date formats |

Table 8.1: Common header files in the C standard library.

## 8.3 The C standard library

The most important library in C programming is called the C standard library. It includes hundreds of functions for doing common operations, such as basic text I/O, file I/O, string manipulation, and mathematical calculations. Its functions include many of the most well known: printf(), strlen(), fopen() and sqrt(). Very few programs are written without making at least some use of this library.

The C standard library is really a collection of libraries that have been grouped together. It makes use of multiple header files (24 as of the 1999 ANSI standard) and multiple library files (depending on system implementation). Because it includes functions covering a wide variety of topics, and because it is organized into multiple files, different parts of it are sometimes referred to in isolation. For example, it is not uncommon to call the math functions portion of the C standard library as simply the "math library". Similarly, it is not uncommon to call the string functions portion as simply the "string library". Table 8.1 lists the most commonly used parts of the C standard library.

Because the C standard library is so commonly used, many compilers simplify the operations required to use it. For example, most C compilers link to the core of the C standard library by default, without requiring the user to specify it. Thus, either of the following lines does the same thing:

```
gcc -o prog1 prog1.c
gcc -o prog1 prog1.c -lc
```

Most compilers include the option **-lc** by default, so that a programmer does not have to type it every time a program is compiled. Some compilers

also include the most common C standard library header files by default. While both of these practices are convenient for experienced programmers, they often confuse novice programmers. The hiding of the basic steps in using a library can cause a novice programmer to make simple mistakes when moving on to additional libraries.

One of the most common mistakes is to forget to include a header file. This can lead to some unexpected and often confusing behavior on the part of a program. For example:

```
main()
{
double  a,b;

b=9.0;
a=sqrt(b);
printf("%lf\n",a);
}
```

On some systems, compiling and executing this code may produce the following output:

```
1075970687.000000
```

This of course is not the square root of nine. A novice programmer, upon seeing this, is often confused by the source of the error. Where did the garbage value come from? The answer is that the header file **math.h** was not included, so that the compiler did not know the type of value returned by the sqrt() function. By default, the compiler assumes that all functions return an **int**. However, the sqrt() function actually returns a **double**. This causes a mismatch, where the return value is interpreted erroneously, causing the garbage value to appear.

Some compilers will warn of this potential problem. For example, a compiler may produce the following warning:

```
main.c(8) : 'sqrt' undefined; assuming extern returning int
```

With a little practice, a programmer will come to recognize this type of warning as a potentially serious problem, and try to alleviate its cause. However, some programmers take advantage of the "int-by-default" return value and

code without proper function prototypes or header file usage. An experienced programmer should be aware of this practice, and ready to work with code written in that manner.

## 8.4 The curses library

Curses is a basic graphics library, for use on a character terminal screen. It is the lowest level of graphics, and dates back to the time when most computer displays could only print text (they could not display images or other graphics). These displays were called terminals. Although most modern computer displays can show images and other graphics, the curses library and character graphics in general are still useful. For example, many computing systems use a "boot loader" when first powered. This boot loader runs before the O/S is loaded. Without the O/S, and its device driver used to operate the advanced graphics functions, the computing system is only capable of character graphics. Similarly, when installing an O/S, the more advanced graphics capabilities are typically not yet available. In some embedded systems, a simple character-based display may be all that is required. In all these cases, a library like curses is useful.

The following code serves to demonstrate the basic operations of the curses library:

```
#include <curses.h>

main()
{
initscr();      /* turn on curses */

clear();        /* clear screen */
move(10,20);    /* row 10, column 20 */
addstr("Hello world");  /* add a string */
move(LINES-1,0);        /* move to LL */

refresh();      /* update the screen */
getch();        /* wait for user input */

endwin();       /* turn off curses */
}
```

If this code were stored in a file called **hello.c**, then the following would compile the program[2]:

```
gcc -o hello hello.c -lncurses
```

Note that the header file **curses.h** must be included, and that the library is linked through the command line argument **-lncurses** (ncurses is the "new curses" library, a rewrite of the traditional curses library, and the most current at the time of this writing).

Most graphics libraries use a function to initialize their internal global variables. For example, the library might discover what sort of graphics card or capability the system has, open the device driver for it, and initialize some variables recording its size and other properties. These values will in turn be available to the program using the library through those variables. In the case of curses, the function **initscr()** performs the initialization. After that, the program can access the global variables **COLUMNS** and **LINES** to see the size of the terminal. The program can also access the variable **stdscr** as the default "window", which can be thought of as the library's name for the terminal. The library is closed (the device driver is closed, and any dynamic memory allocated is freed) through the **endwin()** function call. Curses functions cannot be used prior to calling initscr(), and a program should always call endwin() to close out use of the library.

The basic functions in curses are:

```
move(10,20);     /* move cursor to row=10, column=20 */
addstr("Hello");  /* draw string Hello at cursor location */
```

While drawing text, the cursor is moved ahead (incremented one column) per character drawn, similar to standard typing.

### 8.4.1 I/O control

There are three important concepts in I/O control: buffering, echoing, and blocking. This section studies each of these concepts, and shows how they can be implemented using the curses library.

[2]The reader is strongly encouraged to run this program, and all examples in this section, to better learn the concepts.

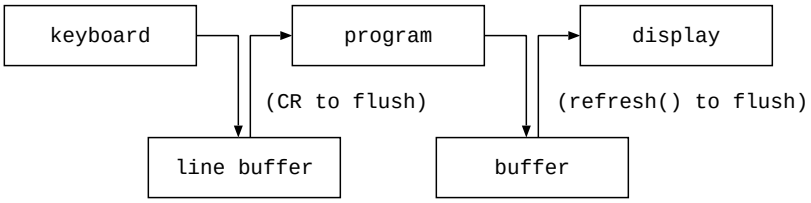


Figure 8.3: Buffering on both the input and output streams.

### Buffering

Buffering refers to the process of temporarily storing bytes on a stream, and grouping them up before transferring them to the destination. Figure 8.3 demonstrates the process. A buffer can be used on any stream; this example shows buffers on both the input and output streams.

By default, characters sent to the curses output window are buffered. This means that the characters are not actually displayed until the buffer is flushed, sending all the characters to the terminal display. Flushing is accomplished by the **refresh()** function call.

Character input is unbuffered by default. This means that functions that read the keyboard (such as **getch()**) return immediately as soon as any key is pressed. This is different from how the C standard input function (**scanf()**) works. The scanf() function is line buffered, meaning that it does not return until the user presses ENTER. The advantage to line buffering is that a user can correct typing mistakes using the backspace or delete key before actually committing the input to the program. These operations are handled by the O/S working on the data in the buffer. Line buffering can be turned on in curses using the **nocbreak()** function. For example:

```
#include <curses.h>

main()
{
initscr();      /* turn on curses */
nocbreak();     /* turn on line buffering */
                /* by default keyboard input is unbuffered */
getch();        /* wait for user input */
endwin();       /* turn off curses */
}
```

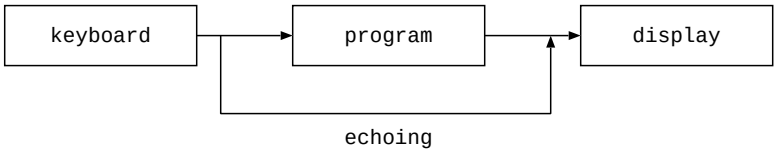


Figure 8.4: Echoing the input stream to the output stream.

Seemingly, this program waits for one character of input from the user, and then terminates. However, when this program is run, the user can type any number of keystrokes; the program will not end until ENTER is pressed. This is because the input is line buffered. Line buffering can be turned off by calling the cbreak() function.

### Echoing

Echoing refers to the process of copying bytes from the input stream to the output stream. Figure 8.4 shows the process. When echoing is turned on, every byte that appears on the input stream is copied directly to the output stream, in addition to being given to the program for processing. Echoing is how a user can see what he or she is typing while providing input to a program.

By default, keyboard input in curses is echoed. The following example demonstrates turning echoing off:

```
#include <curses.h>

main()
{
int     i;
initscr();
noecho();       /* turn off echoing */
for (i=0; i<5; i++)
  getch();      /* wait for user input */
endwin();
}
```

The **noecho()** function call turns echoing off. When this program is run, characters typed are not seen on the screen. Echoing can be turned back on using the **echo()** function.

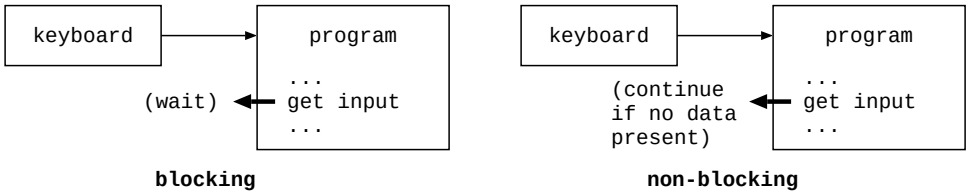


Figure 8.5: Types of blocking on an input stream.

### Blocking

Blocking refers to the process of how the program will wait for bytes to appear on the input stream. Figure 8.5 shows the process. When blocking is turned on, every function call for input will wait until data appears on the input stream. The program will not continue until input is received. When blocking is turned off, the function call will check to see if data is present. If data is present, the read occurs normally, just as if blocking was turned on. However, if no data is present, the function call will return immediately and inform the program that no data was present. This allows the program to continue whether input data is present or not.

By default, the scanf() function is a blocking function; it will wait until input is received. The same is true of the curses library. The following example demonstrates turning blocking off:

```
#include <curses.h>

main()
{
int     i;
initscr();
nodelay(stdscr,TRUE);   /* turn off blocking */
for (i=0; i<5; i++)
  {
  getch();      /* wait for user input? */
  sleep(1);
  }
endwin();
}
```

When this program is run, even if the user does not touch the keyboard, the

program finishes in 5 seconds.

Blocking does not have to be on (indefinite) or off (immediate). Blocking can occur for a pre-selected amount of time, allowing the program to continue if no input is received during that time. The following example demonstrates timed blocking:

```
#include <curses.h>

main()
{
int     i;
initscr();
halfdelay(5);   /* blocking = 5/10 second */
for (i=0; i<5; i++)
  getch();      /* wait for user input */
endwin();
}
```

In this example, blocking is set to 0.5 seconds. Each getch() function call will wait 0.5 seconds for input to appear, but if nothing appears in that time, the function returns and the program continues. Thus, if this program is run without touching the keyboard, it will run for 2.5 seconds and then end.

Being able to control blocking is important in a number of situations. For example, a banking machine typically does not wait forever for a user to provide or password, or to command a transaction. After waiting a fixed amount of time, a banking machine program typically continues to a portion of the program that ends the banking session, in order to protect the user. This is an example of timed blocking. For another example, consider a movie playing program. A user expects to be able to rewind, pause, and interact with a film while it is playing back. This can be accomplished through non-blocking input. This will be explored more in the next section.

### 8.4.2 Dynamic graphics

The curses library is most commonly used for menus and basic user interaction. However, it can be used to create dynamic or moving graphics, albeit of a limited nature. In this section we study a few techniques to create dynamic graphics. Although the techniques are demonstrated using the curses library, the same techniques can be applied using other graphics libraries.

### Motion

The most basic technique to creating a moving graphic is to erase the screen at the graphic's previous location, immediately redrawing it at an adjacent location. Repeating this process over and over provides the illusion of motion. The following code demonstrates the technique:

```
#include <curses.h>

main()
{
int     i;

initscr();
clear();        /* clear screen */

for (i=0; i<30; i++)
  {
  move (10,i);
  addstr("Hello world");
  refresh();               /* flush buffer */
  usleep(100000);          /* pause 0.1 seconds */
  move (10,i);             /* back to previous spot */
  addstr("           ");  /* draw empty space */
  }
getch();
endwin();
}
```

Running this program, the user will see the phrase "Hello world" move horizontally across the screen. The **usleep()** function call is used to control the rate of motion. The usleep() function pauses the program for the given number of microseconds, allowing for finer control of pausing as compared to the sleep() function. Note that the output buffer must be flushed (using the refresh() function) at the appropriate time, or the technique will not work. If the refresh() function call were moved to the bottom of the loop, then the graphic would never be seen. The same is true of the pause (using the usleep() function). The correct order of operation is (1) draw the graphic, (2) flush the buffer, (3) pause the program, (4) erase the graphic, and (5) move

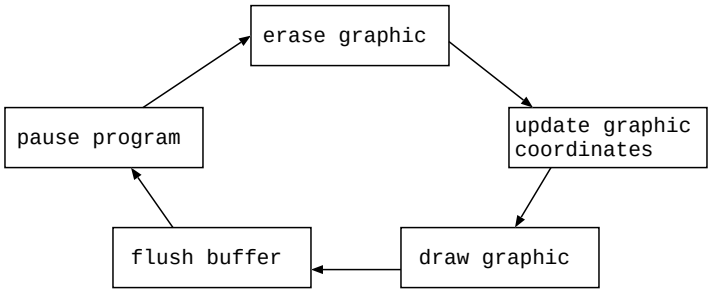


Figure 8.6: Steps in creating a moving graphic. It is often convenient to start the loop with the "erase graphic" step.

to a new location. Figure 8.6 shows the process. Because it is cyclical, it can be started at any point. It is often convenient to put all the code involving a single graphic together, with the flush and pause at the end. In this case, the loop would start with the step that erases the graphic. We will see this again below.

### User input during motion

Moving graphics generally require loops, as described above. The graphic stays in motion only so long as the loop keeps iterating. If we require the user to be able to provide input to the program, while the graphic is in motion, then we must turn blocking off. Otherwise, any function call for input will wait until input is received. Meanwhile, the motion of the graphic will seem to pause. Using a fixed-time blocking is also generally a bad idea, unless the fixed-time is very small. Otherwise, during iterations where the user immediately provides input, the graphic will move faster than during iterations where the program waits for the blocking to time out.

The following code demonstrates *polling* for user input while a graphic is in motion. Polling refers to the process of using a non-blocking function call to check for user input, and acting upon the input if given, but otherwise continuing program execution.

```
#include <curses.h>

main()
{
```

```
int     i,row;
char    ch;

initscr();
clear();
nodelay(stdscr,TRUE);   /* turn off blocking */
row=10;

for (i=0; i<30; i++)
  {
  move (row,i);
  addstr("Hello world");
  refresh();
  usleep(100000);
  move (row,i);
  addstr("           ");
  ch=getch();           /* poll for input */
  if (ch == 'z')        /* act on input */
    row++;
  }
getch();
endwin();
}
```

This program works similarly to the last example, but if the user presses 'z' then the graphic will move down a line. With or without input, the graphic will continually move rightwards across the screen. Executing this program, the user will notice that when the input 'z' is given, it is also displayed on the screen, near the moving graphic. This is a consequence of echoing. In most programs using dynamic graphics, echoing is turned off. This can easily be added to this example by calling noecho() before the loop starts.

### Varying-rate graphics

Using the basic loop structure outlined above, graphics move at the rate defined by the amount of time spent paused in each iteration. Increasing the usleep() causes the graphics to move slower, while decreasing the usleep() causes the graphics to most faster. However, if multiple graphics are dis-

played, they would all move at the same rate. How can different graphics be moved at different rates?

One answer is to use modulus arithmetic on the loop counter to control when the motion of each graphic occurs. If one graphic moves every iteration, but another graphic only moves every other iteration, then the first graphic is moving twice as fast as the second. The following code demonstrates this technique:

```
#include <curses.h>

main()
{
int     i;

initscr();
clear();

for (i=0; i<30; i++)
  {
  move (10,i);
  addstr("     ");
  move (10,i+1);
  addstr("Hello");
  if (i%2 == 0)  /* every 2nd iteration */
    {
    move (12,i/2);
    addstr("     ");
    move (12,i/2+1);
    addstr("world");
    }
  move (LINES-1,0);
  refresh();
  usleep(100000);
  }
getch();
endwin();
}
```

As mentioned previously, it is convenient to start the loop with the step that

erases the graphic. In this way, all the code involving a single graphic can be grouped together, and the flush and pause happen at the end of the loop. The line **move(LINES-1,0);** puts the cursor at the bottom left corner of the screen, so that it does not bounce around following the graphics as they move.

## 8.5 The X library

In order to understand the X library, we first describe how grahpics libraries in general have developed. It is beneficial to examine the graphics libraries used on both a linux/unix system and an MS Windows system. There are many similarities, and some differences, which help to highlight things a system programmer needs to know.

In Section 8.2, we saw that a graphics library serves as a standarized set of function calls between an operating system (in particular, a device driver) and an application. This is so that the same application can work with different graphics displays with varying capabilities. Recent times have seen a tremendous explosion in these capabilities, from simple 2D raster buffering to full texturing, lighting control, and complex rendering of 3D objects and scenes. Applications depend upon graphics libraries to implement all of these capabilities, either in hardware (if supported by the available graphics hardware) or in software in the library itself.

As graphics hardware capabilities have expanded, a hierarchy of graphics libraries has evolved that somewhat resembles the progression of capabilities. Figure 8.7 shows this hierarchy and some of the popular graphics libraries. On a linux/unix system, the X library is at the bottom level. The X library provides for the creation and manipulation of windows. Each window can serve as a separate "screen" or "display". This allows a user to run a number of different applications at the same time, each having its own graphical display, even though the system itself has only one monitor or hardware display. This capability has become so commonplace that users of desktop computers expect it by default. However, it would not be available without the X library or an equivalent.

In addition to windows functions, the X library provides functions for drawing simple 2D graphics, such as lines, circles, and rasters (images). The basic properties of these primitives can be manipulated, such as line color, width, and type (e.g. dotted, dashed or solid). The X library also provides

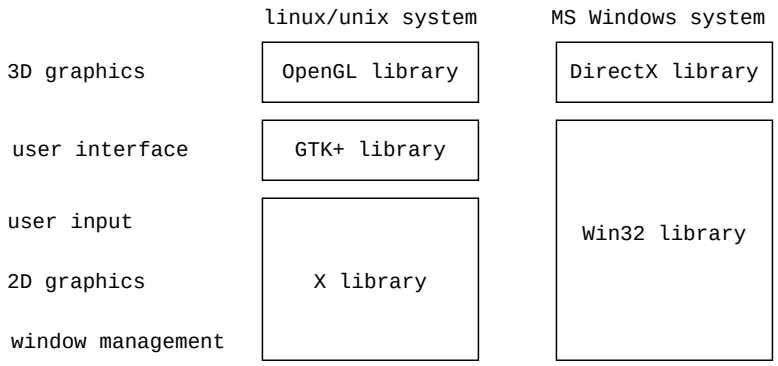


Figure 8.7: Hierarchy and content of popular graphics libraries.

functions for interacting with user input devices, particularly a keyboard and a mouse. Most importantly, input can be directed to the appropriate window (and program) depending on how the user is interacting with the overall system.

The equivalent library on an MS Windows system is the Win32 library. It provides all the above-described capabilities: window creation and manipulation, drawing of 2D graphics, and control of user input. However, it provides an additional capability not available in the X library: the *user interface*. On an MS Windows system, all menus look and operate similarly. All dialog and message boxes look and operate similarly. When opening a file, the interface looks similar from application to application. This is because the Win32 library provides a set of functions to create and interact with menus, dialog boxes, message boxes, and other aspects of a standard user interface. The X library does not provide an equivalent set of functions. The developers of the X library wanted all parts of the system to remain modular. Any system operator is free to install and set up the user interface of his or her choice. Since there are no functions available in the X library for a standard user interface, additional libraries have been developed that provide different user interfaces. These libraries include GTK+, Motif, and Qt. While this initially seemed like a good idea, and in the spirit of modular system development, it turned out to be problematic. Only a small percentage of computer users want the capability to change the standard user interface. Most application developers rely upon a standard user interface. Even experienced system programmers typically prefer to rely upon a standard user interface. Many users take advantage of the capability to fine-tune or adjust a user interface

to suit individual preferences, but there does not seem to be any advantage to providing completely unique user interfaces to all users. At the time of this writing, the GTK+ library is a popular user interface library on a linux/unix system, but there is no consensus standard.

Another important difference between the X library and the Win32 library is the separation of "display" from the hardware. Using the X library, an application can open a window and interact with a user on hardware that is separate from the hardware on which the application is running. This is accomplished through networking. For example, a user can remotely log into a machine, run an application on that machine, and yet graphically display its output on the local monitor. This process can also be done in reverse, displaying output on a remote machine. The Win32 library does not provide for separation of window and display. The library can only interact with hardware directly connected to the system. While this capability can sometimes be useful, it is rarely used by system programmers. As emphasized throughout this book, experienced system programmers prefer to interact with programs through a shell interface. A remote text-display capability is usually sufficient for interacting with programs through a network.

As 3D graphics have become popular, particularly for games, another level of libraries has developed to satisfy the need for hardware-independent application development. On a linux/unix system, the OpenGL library is commonly used. It provides functions for mainpulating and rendering 3D meshes, applying textures, and controlling lighting. On an MS Windows system, the DirectX library is commonly used. It provides a similar set of functions for manipulating and displaying 3D graphics. Although primarily developed for specific systems, there are implementations of both libraries available for other platforms. The main difference is that the OpenGL library is open source while the DirectX library is proprietary. At the time of this writing, both are available for free.

There is one last important difference to discuss between a linux/unix system and an MS Windows system. On an MS Windows system, not only is the user interface standardized, but the system interface is also standardized. All windows look and operate similarly. The titlebar is a standard size with a standard font. The application menu always list from the top left corner, rightwards, and each is a pull-down menu. The system menu for each window always appears in the top right corner and consists of three consistent icons: lower horizontal line (minimize window), square (maximize window), and X (close window). The mouse always uses the same cursor. The overall system

menu is always in the bottom left corner, and the clock is always displayed in the bottom right corner. Right clicking on the background (or desktop) brings up a menu to control desktop appearance, while left clicking on the desktop allows the user to drag icons or start applications. On a linux/unix system, the system interface is not standardized. The gnome desktop is popular, but at the time of this writing there is no consensus standard.

The lack of standardization on linux/unix systems is a result of modular development. This allows users maximum flexibility, and to some degree provides for more consistent interfacing between the various parts of the system. The system software for an MS Windows system is largely monolithic and integrated. The advantage to this approach is that a user can expect a system and its applications to appear and operate in a somewhat predictable manner. This typically decreases the time necessary for a user to become proficient with a new computer, or even a new application. The user does not need to spend time becoming familiar with new icons, menu operations or placements, or appearances. This is one of the strengths of an MS Windows system; it helps allow relatively inexperienced users to operate the system. At the time of this writing, there is growing momentum towards standardizing the linux user and system interfaces. These decisions of course have direct impact upon the libraries a system programmer can expect to use.

### 8.5.1 Windows

The basic construct in X library programming is a **window**. A window is a virtual monitor or display created for a program. It allows multiple programs to operate sharing the same physical monitor or display, each using its own window. In this paradigm, a program must create and manage a window where the output will be displayed. The window also controls input to the program. Typically, a system will only send keyboard and mouse input to a program when the program's window is *active*. Commonly, a window is active when the user selects the window by clicking on it, or when the user moves the mouse into the on-screen area of the window. Window activation depends upon the particular system interface.

The following code demonstrates the basic steps involved in a program using the X library to create a window:

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

```
  #include <X11/Xlib.h>         /* X library definitions */

  main(int argc, char *argv[])
  {
  Display       *Monitor;       /* screen to display on */
  Window        DrawWindow;     /* the window to be created */
  GC            WindowGC;       /* graphics context */


        /* First, every X program must connect to a display */
  Monitor=XOpenDisplay(NULL);
  if (Monitor == NULL)
    {
    printf("Unable to open graphics display\n");
    exit(0);
    }

        /* Create a window - describe a few attributes */
  DrawWindow=XCreateSimpleWindow(Monitor,RootWindow(Monitor,0),
10,10,                      /* x,y on screen */
100,50,                     /* width, height */
2,                          /* border width */
BlackPixel(Monitor,0),      /* foreground color */
WhitePixel(Monitor,0));     /* background color */

        /* Create a default graphics context */
  WindowGC=XCreateGC(Monitor,DrawWindow,0,NULL);

        /* Place the window onscreen, and flush buffer */
  XMapWindow(Monitor,DrawWindow);
  XFlush(Monitor);

        /* wait 2 seconds, then close X library */
  sleep(2);
  XCloseDisplay(Monitor);
  }
```

Assuming this code is stored in a file called **window.c**, then the following

command compiles the code:

```
gcc -o window window.c -L/usr/X11R6/lib -lX11
```

The **-L/usr/X11R6/lib** command tells the compiler to search in the path /usr/X11R6/lib for additional library files. Depending on how the compiler is configured, this option may or may not be necessary (the compiler may already have that path added to its default list of places to look for linking to library files). The command **-lX11** tells the compiler to link to the X11 library file.

In the example, the first function called is **XOpenDisplay()**, which initializes the library for use by the program. It also creates a connection to a physical display (this example uses the default display), discovering its properties and using them to initialize the library. The second function called is **XCreateSimpleWindow()**, which creates a window for use by the program. There are several function calls that create a window, with varying degrees of control over the window's appearance. This one is the simplest. The third function called is **XCreateGC()**, which creates a **graphics context**. A graphics context contains information about how graphics should be drawn in the window. This information includes things like what font to use, how thick to draw lines and other primitives, and what color to use when drawing. This example demonstrate creating a graphics context having all default values. The **XMapWindow()** function call draws the window on the given display (remember that in the X library, the concepts of window and display are separated, so that a window can be drawn on multiple different displays). Since the output display is buffered, the **XFlush()** function call is needed to force flushing of the buffer, to insure that the window actually appears on-screen. Finally, the example program sleeps for two seconds and then closes its use of the X library.

Both **Window** and **GC** (graphics context) variables are actually structures. Each contains a list of variables, the former about how the window appears, and the latter about how to draw into the window. A program can create any number of windows and graphics contexts, each having a different variable name. It is possible to use a single graphics context for all windows.

When running this program, it is possible that the window will not appear at the specified location (10,10). This is due to the involvement of a **window manager**. A window manager is a program that runs on the system and actually controls the placement of windows. It typically tries to

place windows so that they all have minimal overlap. It may therefore override a program's request for a specific window location, in favor of another position. There are function calls in the X library that can override the window manager, and force placement of the window according to the program's specifications, such as **XSetWMHints()**. These functions are beyond the scope of this text.

### 8.5.2 2D graphics

There are a large number of functions in the X library that draw 2D graphics. The following code demonstrates the drawing of a line:

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

main(int argc, char *argv[])
{
Display       *Monitor;
Window        DrawWindow;
GC            WindowGC;
int           x1,y1,x2,y2;

Monitor=XOpenDisplay(NULL);
DrawWindow=XCreateSimpleWindow(Monitor,RootWindow(Monitor,0),
      10,10,          /* x,y on screen */
      100,50,         /* width, height */
      2,              /* border width */
      BlackPixel(Monitor,0),
      WhitePixel(Monitor,0));
WindowGC=XCreateGC(Monitor,DrawWindow,0,NULL);
XMapWindow(Monitor,DrawWindow);
XFlush(Monitor);

while (1)
  {
  printf("Line coordinates? ");
  scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
```

```
    if (x1 == -1)
      break;
    XDrawLine(Monitor,DrawWindow,WindowGC,x1,y1,x2,y2);
    XFlush(Monitor);
    }

  XCloseDisplay(Monitor);
  }
```

After initializing the library and creating and mapping a window, this program goes into a loop. The loop uses the traditional printf() and scanf() functions to get the desired endpoints of the line from the user. It then calls **XDrawLine()** with the given coordinates. The origin of the coordinate system is the top left, with the x-axis positive rightwards and the y-axis positive downwards. Units are pixels; for reference, the window created in this example is $100 \times 50$ pixels in size.

Additional functions for drawing 2D graphics include **XDrawRectangle**, **XDrawPoint**, and **XDrawArc**. The latter can be used to draw a circle, an ellipse, or any portion of an arc.

### 8.5.3 Graphics properties

The properties controlling how graphics are drawn are stored in the graphics context (GC). The Win32 library has a similar construct called a device context (DC). The following code demonstrates changing the color of the lines drawn from the default (black) to blue:

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

main(int argc, char *argv[])
{
Display               *Monitor;
Window                DrawWindow;
GC                    WindowGC;
int                   x1,y1,x2,y2;
XGCValues             GCValues;
```

```
unsigned long         GCmask;
int                   i;

Monitor=XOpenDisplay(NULL);
DrawWindow=XCreateSimpleWindow(Monitor,RootWindow(Monitor,0),
      10,10,
      100,50,
      2,
      BlackPixel(Monitor,0),
      WhitePixel(Monitor,0));
WindowGC=XCreateGC(Monitor,DrawWindow,0,NULL);
XMapWindow(Monitor,DrawWindow);
XFlush(Monitor);

      /* change the foreground color to blue */
GCmask=GCForeground;
GCValues.foreground=0x0000FF; /* red is 0xFF0000 .... */
i=XChangeGC(Monitor,WindowGC,GCmask,&GCValues);
if (i == 0)
  {
  printf("Unable to change GC values\n");
  exit(1);
  }

while (1)
  {
  printf("Line coordinates? ");
  scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
  if (x1 == -1)
    break;
  XDrawLine(Monitor,DrawWindow,WindowGC,x1,y1,x2,y2);
  XFlush(Monitor);
  }

XCloseDisplay(Monitor);
}
```

The **XChangeGC()** function call takes in three relevant arguments: the GC in which values are to be changed, a new set of values, and a mask. The new set of values is stored in an **XGCValues** variable, which is another structure. The mask variable indicates which values in that structure are to be used to change the given GC. Multiple values can be changed in a single XChangeGC function call. For example:

```
GCmask=GCForeground | GCLineStyle | GCLineWidth;
GCValues.foreground=0x0000FF;
GCValues.line_style=LineDoubleDash;
GCValues.line_width=4;
      /* man XChangeGC to see all GC properties */
XChangeGC(Monitor,WindowGC,GCmask,&GCValues);
```

This code changes the foreground color, the line style, and the line width. The man page for XChangeGC() or a similar reference can be used to see all the properties that are changeable in a graphics context.

### 8.5.4 User input

Using the X library, user input is provided to a program through **events**. An event occurs every time the user manipulates an input device. This includes key presses, key releases, mouse motion, and mouse button presses and releases. An event can also be generated by the operating system in response to actions taken by another program. For example, if a program ends, destroying its window and thereby uncovering another window, the operating system will send an event to the program associated with the newly uncovered window.

The following code demonstrates using events to obtain user input:

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

main(int argc, char *argv[])
{
Display       *Monitor;
Window        DrawWindow;
```

```
GC              WindowGC;
XEvent          SomeEvent;
long int        EventMask;

Monitor=XOpenDisplay(NULL);
DrawWindow=XCreateSimpleWindow(Monitor,RootWindow(Monitor,0),
      10,10, 100,50, 2,
      BlackPixel(Monitor,0),
      WhitePixel(Monitor,0));
WindowGC=XCreateGC(Monitor,DrawWindow,0,NULL);
XMapWindow(Monitor,DrawWindow);
XFlush(Monitor);

      /* Tell X server which events to pass to program */
EventMask=ButtonPressMask;
XSelectInput(Monitor,DrawWindow,EventMask);

while (1)
  {
  XNextEvent(Monitor,&SomeEvent);  /* get user input */
  if (SomeEvent.type == ButtonPress)
    printf("Button pressed!\n");
  }

XCloseDisplay(Monitor);
}
```

In this example, if a mouse button is pressed, then the program prints out a message to the user. The **XSelectInput()** function call tells the system which events the program is interested in receiving. For example, a program may only use the mouse, and so would not include keyboard-related events in its event mask. The **XNextEvent()** function can be used to obtain input from the user. Once it returns, a program can decide what to do with the given event.

There are several functions that vary in how events are received by a program. The XNextEvent() function is a blocking function; it will not return until an event has been received. The **XPeekEvent()** function can be used with appropriate coding to implement non-blocking input polling.

A program can request multiple types of events be sent to it, and then process them differently. For example:

```
EventMask=ButtonPressMask | KeyPressMask | PointerMotionMask;
      /* see /usr/include/X11/X.h for list of all masks */
XSelectInput(Monitor,DrawWindow,EventMask);

while (1)
  {
  XNextEvent(Monitor,&SomeEvent);
      /* man XEvent, and its derivatives (e.g. XButtonEvent)
      ** for complete lists of event types and contents */
  if (SomeEvent.type == ButtonPress)
    printf("Button pressed!\n");
  if (SomeEvent.type == KeyPress)
    printf("Key pressed!\n");
  if (SomeEvent.type == MotionNotify)
    printf("Mouse is moving!\n");
  }
```

Using these concepts, we can use mouse input to control the drawing of lines. The following code can replace the text-based interface from the example in Section 8.5.2:

```
int    WhichPoint;

WhichPoint=0; /* 0=>first point, 1=>second point */
while (1)
  {
  XNextEvent(Monitor,&SomeEvent);
  if (SomeEvent.type == ButtonPress)
    {
    if (WhichPoint == 0)
      {
      x1=SomeEvent.xbutton.x;
      y1=SomeEvent.xbutton.y;
      WhichPoint=1;
      }
```

```
    else
      {
      x2=SomeEvent.xbutton.x;
      y2=SomeEvent.xbutton.y;
      WhichPoint=0;
      if (x1 == x2  &&  y1 == y2)
        break;    /* exit loop and program */
      XDrawLine(Monitor,DrawWindow,WindowGC,x1,y1,x2,y2);
      XFlush(Monitor);
      }
    }
  }
```

### 8.5.5 Fonts

A shell display uses a fixed grid of character graphics. Characters can only be drawn inside the grid cells. For example, a character cannot be drawn halfway between two lines of text. In addition, the font is fixed and is typically courier, where every character fills the same amount of space. With the X library, a program can draw text at any location in a window, using any font. There is no character grid, instead the units of location are pixels. In order to draw text using a specific font, a program must first set up the graphics context to know how to draw with that font. The following code demonstrates using the X library to draw text:

```
#include <stdio.h>
#include <string.h>
#include <X11/Xlib.h>

main(int argc, char *argv[])
{
Display         *Monitor;
Window          DrawWindow;
GC              WindowGC;
int             x1,y1;
XGCValues       GCValues;
unsigned long   GCmask;
XEvent          SomeEvent;
```

```
char            text[80];
Font            NewFont;

Monitor=XOpenDisplay(NULL);
DrawWindow=XCreateSimpleWindow(Monitor,RootWindow(Monitor,0),
        10,10, 100,50, 2,
        BlackPixel(Monitor,0),
        WhitePixel(Monitor,0));
WindowGC=XCreateGC(Monitor,DrawWindow,0,NULL);
XMapWindow(Monitor,DrawWindow);
XFlush(Monitor);

NewFont=XLoadFont(Monitor,"r14");

GCmask=GCForeground | GCFont;
GCValues.foreground=0xFF0000;
GCValues.font=NewFont;
XChangeGC(Monitor,WindowGC,GCmask,&GCValues);

XSelectInput(Monitor,DrawWindow,ButtonPressMask);

while (1)
  {
  XNextEvent(Monitor,&SomeEvent);
  if (SomeEvent.type == ButtonPress)
    {
    x1=SomeEvent.xbutton.x;
    y1=SomeEvent.xbutton.y;
    strcpy(text,"Hello!");
    XDrawString(Monitor,DrawWindow,WindowGC,x1,y1,
              text,strlen(text));
    XFlush(Monitor);
    }
  }

XCloseDisplay(Monitor);
}
```

This program will draw "Hello!" at the current mouse location whenver the user presses a mouse button. The program uses the **XLoadFont()** function to load information from the system about the "r14" font. It then assigns that font to the graphics context for the created window. The program uses the **XDrawString()** function to actually draw the text. The properties of the text are controlled by the values previously set in the graphics context.

The information about how to draw text using a particlar font is stored in a font file on the system. Font files use a variety of formats, but they all contain the same basic information: the appearance of all characters in the given font. In order to use a font, a program must identify that font by its name. The X library provides functions to identify the fonts available on a system. The following code demonstrates these functions:

```
#include <stdio.h>
#include <X11/Xlib.h>

main()
{
char            text[80],partial[80];
char            **AvailableFonts;
int             font_count,i;
Display         *Monitor;

Monitor=XOpenDisplay(NULL);
printf("Enter a string to search: ");
scanf("%s",partial);
sprintf(text,"*%s*",partial);
AvailableFonts=XListFonts(Monitor,text,10,&font_count);
for (i=0; i<font_count; i++)
  printf("%s\n",AvailableFonts[i]);
XFreeFontNames(AvailableFonts);
XCloseDisplay(Monitor);
}
```

The **XListFonts()** function searches the system for fonts matching the given string, and returns a list of all font names that partially match. The example program asks for at most 10 matches, and prints them out. The **XFreeFontNames()** function should be called to free up the memory allocated for

storing the font names in the XListFonts() function. There are additional functions related to loading and handling fonts; these are beyond the scope of this text.

## 8.6 Library pitfalls

Once a programmer gets used to the idea of using libraries, it is easy to get enthralled by them. They save time, and allow us to code things that might otherwise be very difficult. It is important to remember that a library is just a tool. Libraries should be used to help overcome problems, not just for the sake of their existence. A programmer can make the mistake of using a library when its utility to a given problem is minimal. This is bad, because now an application is tied to a library that it doesn't really need. Programs can become bloated and difficult to maintain simply because they have been linked to too many libraries.

Another common pitfall is to spend too much time looking for a library to solve a problem. It can be enticing to think that somebody "out there somewhere" has already written code to tackle the problem at hand. This leads to a programmer unwilling to solve the given problem from scratch, instead searching for library-supported existing solutions. This can end up taking more time in the search than it would to simply write code from scratch. It can also cause a programmer to use a library that does not quite fit the problem at hand, but can be forced to provide a hacked solution. This leads to inefficient and sometimes error-prone applications.

A library is just another tool in the arsenal of an experienced system programmer. Like a debugger, or a shell, or a system call, it is there to help solve problems. A carpenter may use a hammer, wrench, or screwdriver to work on something, but (hopefully) only in the appropriate circumstances. Similarly, a programmer should only use the tools at hand when the job calls for them.