LIBRARIES

-A library is a set of functions, packaged as a system resource and intended for use by other programs on the system.
A library is not an executable program therefore it does not include the main function..
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 10 or so functions, while other libraries may contain a thousand functions. Some are considered standard, having become common to a large number of computing systems.
Examples include the C standard library and the X library.
When using a library, a program need not 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 the latter 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.
Advantages of using libraries:

i. Using a library saves time in programming, because a programmer can make use of existing code. Library code tends to be written by experts and thus tends to have good design and performance.
ii. 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.

The C standard library provides hundreds of functions for common text processing and mathematical operations.
The curses library provides functions for creating a character-based graphical interface.
The X library provides functions for a pixel-based graphical interface, including windows and mouse interaction.

Using a Library

There are two basic steps to using a library

i. One or more header files must be included in the C program code
ii. 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.

Header Files

We include the header file as shown in the following example:

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

It 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.

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, including 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:

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

This prototype tells us that 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 */
```

This provides a constant value for pi.

A header file may use typedef and struct definitions to create library-specific aliases for common data types or to create 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.

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.
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 nonstandard 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.

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. On a Unix system, library files are typically stored in /usr/lib.
-Unix 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 these 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
```

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

Purpose of Libraries

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

1) **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.
2) **Difficult to code.** An example in this category is the math library. The functions in the 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 multiply 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 have already 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.
3) **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. 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 developing an application only 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.

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 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.

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 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." The table below summarizes the most commonly used parts of the C standard library

Common header files in the C standard 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

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.

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 9.
This error is caused by 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 cause 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
```