C Libraries

In general, libraries are created from many library source files, and are either built as archive files (libmine.a) that are statically linked into executables that use them, or as shared object files (libmine.so) that are dynamically linked into executables that use them. To link in libraries of these types, use the gcc command line options -L for the path to the library files and -l to link in a library (a .so or a .a):

```
    -L{path to file containing library} -l${library name}
```

For example, if I have a library named libmine.so in /home/newhall/lib/ then I'd do the following to link it into my program:

```
   $ gcc -o myprog myprog.c  -L/home/newhall/lib -lmine
```

You may also need to specify and include path so the compiler can find the library header file: -I /home/newhall/include

If you create your own shared object files and do not install them in /usr/lib, then you need to set your LD_LIBRARY_PATH environment variable so that the runtime linker can find them and load them at run time. For example, if I put my .so files in a directory named lib in my home directory, I'd set my LD_LIBRARY_PATH enviroment to the following:

```
  # if running bash:
  export LD_LIBRARY_PATH=/home/newhall/lib:$LD_LIBRARY_PATH

  # if running tcsh:
  setenv LD_LIBRARY_PATH /home/newhall/lib:$LD_LIBRARY_PATH
```

USING AND LINKING LIBRARY CODE

```
To use a Library that is not linked into your program automatically by
the compiler, you need to (1) include the library's header file in your
C source file (test.c in the example below), and (2) tell the compiler to
link in the code from the library .o file into your executable file:

    step 1: Add an include line (#include "somelib.h") in a program
            source file (e.g., test.c).

    step 2: Link the program's .c file with the library object file
            (i.e. specify the somelib.o file as a command line argument to gcc):

        % gcc  -o myprog test.c somelib.o

    The resulting executable file (myprog) will contain machine code
    for all the functions defined in test.c plus any mylib library
    functions that are called by
```

CREATING AND USING YOUR OWN LIBRARY CODE

To create a Library of code you need to do the following:

- (1) Create an INTERFACE to your library: mylib.h
- (2) Create an IMPLEMENTATION of your library: mylib.c
- (3) Create a LIBRARY OBJECT FILE that can be linked with programs that want to use our library code
- (3a) or create a SHARED OBJECT FILE from many .o files that can be linked with programs that want to use your library code
- (4) USE the library in other C code: (a) #include "mylib.h" (b) link in the libary code into a.out file
- (5) Set LD_LIBRARY_PATH environment variable for finding shared objects in non-standard locations at runtime

Details:

- (1) INTERFACE: the header file to your library should contain definitions for everything exported by your library:
  - function prototypes with comments for users of your library functions
  - definitions for types and global variables exported by your library

  You should have "boiler plate" code (#ifndef ... #endif) around the header file's contents, to ensures that the preprocessor only includes the mylib.h file one time.

  Here is what an example .h file might look like:

```
#ifndef _MYLIB_H_
#define _MYLIB_H_

    // a constant definition exported by library:
    #define MAX_FOO  20

    // a type definition exported by library:
                struct foo_struct {
        int x;
        float y;
    };
    typedef struct foo_struct foo_struct;

    // a global variable exported by library
    // "extern" means that this is not a variable declaration, it
    // just defines that a variable named total_foo of type int
    // exits and you can use it (its declaration is in some library source file)
    extern int total_foo;
```

```
    // a function prototype for a function exported by library:
    extern int foo(float y, float z);   // a very bad function name

#endif
```

- (2) IMPLEMENTATION: create a mylib.c file that #includes "mylib.h" and contains the implementation of every function in your library.

```
#include "mylib.h"

...
int total_foo;

int foo(float y, float z) {
    ...
}
```

- (3) create a LIBRARY OBJECT FILE that can be linked into other programs that use your library (use the -c option to gcc to tell it just to create an object file (a .o file) rather than an executable:

```
gcc -o mylib.o -c mylib.c
```

  you can then use the mylib.o file as the "library file" and statically link it into other programs that use it, or...

- (3a) alternately, you can create a SHARED OBJECT FILE from one or more .o files that can be linked into other programs that use your library A shared object file is the Unix name for a dynamically linked library whose code is loaded into the a.out file at runtime. To create a .so file use the -shared flag to gcc. Here is what an example build might look like:

```
   gcc -shared -o libmylib.so  mylib.o blah.o grr.o  -lm
```

- (3b) you could also build an ARCHIVE FILE (a statically linked library, libmylib.a) from one or more .o files. If you link with a static library, its code is copied into the a.out file at runtime.

  See gcc documentation for more information on how to build .a and .so files.

- (4) USE the library in other programs:

```
step 1: Add an include line (#include "mylib.h") in all program source files that use library definitions (e.g., test.c).

step 2: Link the program's .c file with the library object file
        (i.e. specify the mylib.o file as a command line argument to gcc):

         gcc  test.c mylib.o

    OR to link in libmylib.so (or libmylib.a):

         gcc  test.c -lmylib

    OR to link with a library not in the standard path:

         gcc  test.c -L/home/newhall/lib -lmylib

    The resulting a.out out will contain machine code for all the functions
    defined in test.c plus any mylib library functions that are called by
    the test.c code.
```

- (5) RUNNING an executable linked with a shared object file:

```
If the shared object file in not in /usr/lib, then you need to set your
LD_LIBRARY_PATH environment variable so that the runtime linker can find
and load your .so file into the executable at runtime:

# in bash:
export LD_LIBRARY_PATH=/home/newhall/lib:$LD_LIBRARY_PATH

# in tcsh:
setenv LD_LIBRARY_PATH /home/newhall/lib:$LD_LIBRARY_PATH
```