

CS220 - Computer System II  
Lab 5

**Due: 09/28/2016, 11:59pm**

## 1 Introduction

Hopefully you are now comfortable with writing programs in C. Write-once-and-reuse-many-times is a valuable practice in programming. **Code libraries** are essential for reusing code. Understanding how libraries work is a key component of C software development. In this lab, you will learn about **two main forms of reusing binary code**, shared and static libraries. You will implement a library that allocates and deallocates memory on the heap. It will print log statements to record all the **allocations and deallocations**. You will write a program to test the shared and static library versions of the library.

## 2 Getting Started

Create a directory called Lab5 in your home directory or any other convenient location. You will implement all the code here.

## 3 Implementing the library

Within Lab5, create `shared.h` with the following contents.

```
1  /* shared.h */
2  #ifndef _SHARED_H
3  #define _SHARED_H
4
5  extern void * logger_malloc (unsigned int size);
6  extern void logger_free (void *p);
7
8  #endif
9
```

The **extern** keyword allows one module of your program to access a global variable or function declared in another module. It tells the compiler that the definition is provided somewhere else.

Next, create `shared.c` with the following contents.

```
1  /* shared.c */
2  #include <stdio.h>
```

```

#include <stdlib.h>
4
void *logger_malloc(unsigned int size)
6 {
    void *ret;
    8 printf("Allocating %u bytes...\n", size);
    ret = malloc(size);
    10 if (ret == 0) {
        printf("Allocation failed! :(\n");
    12 } else {
        printf("Successfully allocated at %p\n", ret);
    14 }
    return ret;
16 }

18 void logger_free(void *p)
{
    20 printf("free()ing memory at %p...", p);
    free(p);
    22 printf("DONE\n");
}

```

The functions `logger_malloc` and `logger_free` are wrappers for `malloc` and `free` that simply print log statements.

## 4 Static library

While using static library, required dependencies are statically linked to the binary during compilation. In other words, the required functions are embedded into the target binary. Static libraries typically have a `.a` extension.

Compile `shared.c` using the following command:

```

1 $ gcc -c -std=c89 -g -Wall shared.c -I$PWD

```

The `-c` option tells `gcc` to simply compile the program and generate an object file `shared.o`, but not generate an executable (an executable can execute, and must contain a main function). In `shared.c`, note that `shared.h` is included as `#include<shared.h>` as opposed

to `#include "shared.h"`. Files, `stdio.h`, `stdlib.h`, etc. are already included in the default search paths. The `-I` option tells gcc additional directories to look for include files in. Therefore, `-$PWD` tells gcc to look for `shared.h` in the current directory.

Next, use the following archiver (`ar`) command to generate an archive of the object file(s) to generate the `.a` file. The archiver is used to package multiple object files into a single file:

```
1 ar rcs libshared.a shared.o
```

▷ Look up the manpage for `ar` to see what options `rcs` do!

Make a directory called `lib` and move `libshared.a` to `lib`.

## 5 Shared library

Shared Libraries are the libraries that can be linked to any program at run-time. They provide a means to use code that can be loaded anywhere in the memory. Once loaded, the shared library code can be used by any number of programs. `libc` is an excellent example of shared library.

Use the following commands to compile `shared.c` and generate the shared library:

```
1 $ gcc -c -std=c89 -g -Wall shared.c -I$PWD
$ gcc -shared -o libshared.so shared.o
```

The `-shared` option tells gcc to create a shared library. `libshared.so` will be the name of the shared library. The shared library must start with `lib` and must have an extension `.so`. Move `libshared.so` to `lib` directory.

## 6 Using the libraries

Create `main.c` with the following contents:

```
2 #include <shared.h>
```

```

1 int main()
4 {
  void *p;
6 p = logger_malloc(10 * sizeof(unsigned int));
  logger_free(p);
8 return 0;
  }

```

**Statically linking libshared.a** Use the following command to statically link libshared.a to main.c.

```

1 $ gcc -std=c89 -g main.c -I$PWD libshared.a -o main_static

```

The library libshared.a is embedded into the resulting `main_static` executable! To test it, extract the disassembly of the main program:

```

1 $ objdump -d main_static > main_static.disas

```

Open it and search for `logger_malloc` and `logger_free`.

▷ Run `main_static` and record the output in `Lab5.txt`. Record the addresses of `logger_malloc` and `logger_free` functions in disassembly of main.

**Dynamically linking libshared.so** Dynamic linking is different when compared to static linking. The dependencies are resolved at runtime. Build main using the following command:

```

1 $ gcc main.c -I$PWD -L$PWD/lib -o main_shared -lshared

```

The `-I` option tells `gcc` where to find `shared.h`. Similarly, the `-L` option tells `gcc` where to look for the shared library. The `-l` option tells `gcc` what the dependent library is. Note that although the name is `libshared.so`, the `-l` option accepts “shared”. The preceding “lib” and the extension are implicit. The information regarding what the dependency is (`libshared.so`) is encoded into the binary (`main_shared`), but not the library itself.

▷ Run `main_shared` and record the output in `Lab5.txt`.

Now, run the following command and see that the shared library location is unknown:

```
1 $ ldd main_shared
```

Now, retry running `main_shared` after setting the `LD_LIBRARY_PATH` variable:

```
1 $ bash
$ export LD_LIBRARY_PATH=$PWD/lib
```

Shared libraries are loaded at runtime. Therefore, at runtime, the loader and dynamic linker (which is different from `gcc`) needs to know where to find the libraries. The variable `LD_LIBRARY_PATH` tells the loader where to look for shared libraries. The `export` command is not supported on the `tee-shell`, which is the default shell on `remote.cs.binghamton.edu`. So, we run `bash` before running `export`.

To test, run the following commands and record the outputs in `Lib5.txt`.

```
1 $ ldd main_shared
2 $ ldd main_static
```

The program `ldd` lists the shared library dependencies of programs.

## 7 Submitting the result

Remove binaries and intermediate files from `Lab5`. Create a `tar.gz` of `Lab5` folder with only `main.c`, `shared.h`, `shared.c` and `Lab5.txt`:

```
$ tar -cvzf lab5\_submission.tar.gz ./Lab5
```

Submit `lab5_submission.tar.gz`.