# CS262, Lab Assignment 11:
# Multiple Source Files

Due: Sunday, May 1st at 11:59 pm ET

## Description:

In this assignment, you will simulate working on a large project containing many different files. Many times, with large projects, source and include files are distributed across several directories. Generally (but not always), source (`*.c`) files are kept in a directory named `src/`, include (`*.h`) files are kept in a directory named `inc/` (or sometimes `include/`), and object files are kept in a directory named `obj/`. The executables are also generally kept in a directory named `bin/`.

You will create several source and include files, place them in the appropriate directories and create a Makefile that will compile and link an executable from these files.

## Background:

Read the following webpages:

https://gcc.gnu.org/onlinedocs/cppinternals/Guard-Macros.html
https://gcc.gnu.org/onlinedocs/gcc-3.3/cpp/Stringification.html

## Instructions:

Perform all the following on Zeus directly rather than doing them on your own computer and then copying them over to Zeus.

- Create a directory named `lab11`
- Inside the `lab11` directory, make the following directories:
  - `src`
  - `inc`
  - `obj`
  - `bin`
  - `lib`
- Download the Makefile, `lab11.h` and `lab11.c` files from Blackboard and place them in their respective directories (The Makefile goes in the `lab11` directory - above the `src/ inc/ obj/` and `bin/` directories)
- Look at the source code for `lab11` and notice the function calls it contains. You are to create a stub source and include file that corresponds to each function call. Name each file after the function call it contains. For example, the code for `function1()` will be found in `Function1.c` and the function prototype will be found in `Function1.h`. The files you create will contain the following:
  Each source file will contain:
  - The standard `#include` statements for `stdio.h` and `stdlib.h`
  - pre-processor macros to include its corresponding header file as well as `lab11.h`
  - The implementation of the stub function corresponding to its (file) name

- Each include file will contain:
  - Guard Macros where the defined macro corresponds to the header file name
    **Example**: for `Function1.h`, the defined macro would be FUNCTION1_H
  - A prototype for the corresponding function found in the source file
- The files created will be placed in the corresponding `src/` and `inc/` directories
- For each source file, place the following code in each function:

```
int i = 0;
   int *p = malloc(sizeof(int) * ARRAY_SIZE);
   if (p == NULL){
      fprintf(stderr, "function1(): Error allocating p\n");
      exit(-1);
   }
   printf("In function1()...\n");
   // Initialize the array
   for (i=0; i<ARRAY_SIZE; i++){
      p[i]=i;
   }
   // Print part of the array
   for (i=0; i<PARTIAL_SIZE; i++){
      printf("function1(): %s = %d,%s = %d ", PR(i),i,PR(p[i]), p[i]);
      printf("\n");
   }
   free(p);
```

- Replace the `function1()` character strings for each code segment with the name of the corresponding function name
- Modify the Makefile, adding information as prompted by the comments in the file.
  Pay careful attention to the comments in the Makefile!
- Compile your program using the make command. Run your program to show that it executes correctly. You should notice a lot of output.

## Testing using Valgrind

Use the `-g` and `-O0` (that's oh-zero) options when compiling with `gcc`.
Your Makefile should contain these options so that it will compile with them automatically

Once you have your program working, use valgrind with the option

```
--leak-check=yes
```

Make sure that the Heap, Leak and Error Summaries using valgrind show that there are no errors or issues.

## Creating a Library

Now that you have the code working with the source, include, object and executable files in their appropriate directories, you will create a C library with which you will compile your code. You have been using C libraries all along when compiling your labs and projects this semester. However, those libraries are system libraries, and are found on all computers that contain a C compiler.

A library is basically an archive of various functions that may be as part of a program's execution. Usually, these functions perform similar tasks or have some aspect about them that requires them to be kept in the same library. An example library would be the math library, where the most common mathematical formulas are gathered.

For this lab, you will create a library containing the object code for all the `functionX()` functions you created earlier in the lab. There are two types of libraries used for compiling C programs, *static* and *dynamic*. For this lab, you will be creating a static library. The command used to create a static library is the ***ar*** (archive) command. Note the similarities between this command and the ***tar*** (tape archive) command.

The ***ar*** command uses files containing object code (`*.o` files) to create a library. These object files have already been compiled, and therefore contain all the information necessary for the linker to use them to create an executable. When the linker creates the executable, it pulls the necessary machine (object) code out of the library, and incorporates it into the executable. To create a static library, the general command is:

```
  ar rc libsomelib.a objectfile1.o objectfile2.o objectfile3.o …
```

Read the `manpage` for ***ar*** for more information.

By convention, static libraries always begin with the three letter "`lib`" prefix, and have a "`.a`" extension.

You should already have compiled object files in the `obj/` directory. You will use them to create a library named "`libLab11.a`" which will be placed in the `lib/` directory. Use the example ***ar*** command given above to create your library and place it in the `lib/` directory. Note that you can use the `*.o` wildcard to use all object files in the obj/ directory. You can create this library directly in the `lib/` directory by changing to the lib directory, and running the ar command, using a relative path to the object directory (…`/obj/*.o`).

After creating the library, you should run a separate command, `ranlib`, to create an index for your library. This is done with the command:

```
  ranlib
```

Try running this command on the library you just created.

## Checking your library:

Once you run `ranlib` on your library, you can use the ***nm*** command to list the index of functions contained in your library. To see if your library was created properly, use the command:

```
 nm libLab11.a
```

It should list the functions that are currently contained in the library.

## Compiling an executable with your library:

When you initially compiled the executable for `lab 11`, you placed all the `*.o` files on the gcc command line. Now that you have a library, you can just use it, along with any other specific source files to create an executable.

Although you used a Makefile to compile your earlier executable, for this next step you will compile your program directly on the command line. To do so, use the `-I` option with gcc, which lets the compiler know where some include (`*.h`) files may be found. You will now also use the `-L` option. This option works in a similar manner to `-I`, but instead lets the compiler know where some libraries (`*.a` files) may be found.

Make sure that you are in the directory that contains all your `bin/ src/ inc/ obj/` and `lib/` directories. You will create a new executable named `lab11_from_lib`. The only source file you will have in your `gcc` command will be `lab11.c`. You will use the following command:

```
gcc -o bin/lab11_from_lib -I./inc src/lab11.c  -L./lib  -llab11
```

Let's break this command down a bit:

| | |
|---|---|
| `-o` | The output file |
| `-I` | The path to the directory containing some header files. This option usually comes before any source files are listed for the gcc command. |
| | The source file being compiled |
| `-L` | The path to the directory containing libraries that will be used during linking |
| `-l` | The name of a library that will be used. Note that with the `-l` option, the three character "`lib`" as well as the "`.a`" extension are not included in the name. |

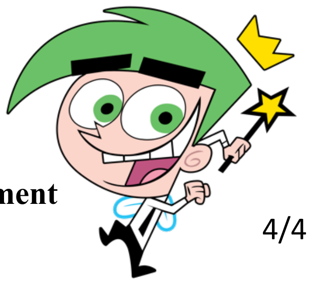
## Why use a library anyway?

There are many reasons why libraries are important to know about when creating large C programs. Libraries are easy, compact ways to give someone access to your source code. They are also a way to "modularize" portions of source code so that similar code is all found in one place.

## Submission:

You will submit a typescript file (showing that your program compiles without errors)

1. Remove all the files contained in the `bin/`, `lib/` and `obj/` directories
2. Create a typescript file named `lab11_typescript_<username>_<labsection>`
3. Show that you are logged onto Zeus
4. List the content of your Makefile
5. List the files in your `src/` and `inc/` directories
6. Show that the `bin/`, `obj/` and `lib/` directories are empty
7. Compile the code using your Makefile
8. Run the code using valgrind: `valgrind --leak-check=yes lab11`
9. Use the gcc command along with your new library to create a separate executable
10. Run the new executable to show that it executes correctly
11. End the typescript
12. Run `make cleanall` to remove your object files as well as the executable
13. Create a `tarfile` of your `lab11_<username>_<labsection>` directory
14. Submit the `tarfile` to Blackboard
15. Verify that your submitted `tarfile` can be extracted and it's the right `tarfile`.



**Congratulations! You have completed your assignment**