Lab 2: Creating Programs with Multiple Source Files

Learning Outcomes

This laboratory tutorial will provide you with the prerequisite knowledge and practice required to develop and implement C programs involving multiple source files. After completing the tutorial, you should be able to demonstrate:

- Familiarity with CSD1121 programming tools and environment.
- Understand the edit-compile-link cycle
- Understand declarations and definitions of functions.
- Understand the process of developing C programs consisting of multiple source files.
- Be able to author simple function definitions.

Prologue

Compiling and linking with a single source file

Open a Window command prompt, change your directory to C:\sandbox [create the directory if it doesn't exist], create a sub-directory lab02, and launch the Linux shell. Download the presentation deck L4+intro+to+c+part2+presentation.pdf from Lecture 4. Using Code, create a new source file called hello.c with the exact code shown on slide 33. Notice that source file hello.c contains a line starting with the pound sign `#`. This line specifies a preprocessor directive. A *preprocessor* is a text-editing program invoked by the compiler toolchain to prepare your source file for compilation. *Preprocessor directives* specify the editing actions the preprocessor must perform. All preprocessor directives begin with a pound sign `#`. Ritchie could have chosen any of several different symbols to identify preprocessor directives; he chose the pound sign. This is just one of the rules of C known as its **syntax**. Preprocessor directives can start in any column but they traditionally start in column 1. One of the most important of the preprocessor directives, and one that is used in virtually all programs, is `include`. The `include` directive tells the preprocessor that you need information from selected libraries or files known as header files. In today's complex programming environments, it is almost impossible to write even the smallest of programs without at least one library system. In source file hello.c, you're printing some text to the console using C standard library function `printf`

```
1 printf("Hello World\n");
```

and therefore you'll need to include the standard input/output header file stdio.h. The complete syntax for the directive to include the standard input/output header file is shown below.

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

The format, or syntax, of this directive must be exact. Since it is a preprocessor directive, it starts with the pound sign. There can be no space between the pound sign and keyword `include`. The keyword `include` means just what you would think it does. It tells the preprocessor that you want the header file in the angular brackets (`<` `>`) added to your source file. The name of the header

file is stdio.h. This is an abbreviation for *standard input/output*. The suffix .h is short for **header file** and is a traditional C/C++ convention. The angular brackets specify that this is a C standard library header file and the preprocessor will fetch the file from the installation directory of the compiler. The GCC manual provides the list of directories searched by gcc to find header file stdio.h.

What does header file stdio.h contain that requires it to be included in source file hello.c? Recall that C/C++ compilers require every name or identifier used in a source file to be declared [who am I?] before the first use of the name. Header file stdio.h contains a declaration of function `printf`. The insertion of this declaration by the preprocessor in the source file allows allows the compiler to know the meaning and intent of the object associated with that identifier and ensure that the object is used correctly in the source file. You can visually see the declaration of function `printf` by asking the compiler toolchain to stop the compilation process right after the execution of the preprocessor:

```
gcc -std=c11 -Wall -Wextra -Werror -E hello.c -o hello.i
```

The above command will include the contents of header file stdio.h in place of the line `#include <stdio.h>` and the edited contents are written to a new file hello.i. Open this new file in your text editor and search for the declaration of function `printf` which will look like this:

```
extern int printf (const char *__restrict __format, ...);
```

You can then run the compiler proper on the preprocessor file hello.i to create the assembly output like this:

```
gcc -std=c11 -Wall -Wextra -Werror -S hello.i -o hello.s
```

Confirm that output file hello.s contains assembly code. The next step in the compiler toolchain involves the conversion of the assembly file hello.s to an object file hello.o:

```
gcc -c hello.s -o hello.o
```

The object file hello.o contains machine code in the form of strings composed of 0s and 1s and therefore cannot be read by a text editor. The final step in creating an executable is to link the object file hello.o with the C standard library object file:

```
gcc hello.o -o hello.out
```

To understand the linking process, you must understand that hello.o doesn't contain the definition [or implementation] of C standard library function `printf`. The compiler compiled your source file hello.c without access to the definition of function `printf`; instead, it just had the declaration of the function. The definition of C standard library function `printf` is part of the installation of GCC and is located in an archive file [think of an archive file as just a collection of object files] libc.a whose location on your machine is determined by running the command:

```
gcc -print-file-name=libc.a
```

The purpose of linking is to combine your object file hello.o [containing the definition of function `main`] and archive file libc.a [which contains the definition of function `printf`] into a single executable file hello.out that contains the definitions of both functions `printf` and `main`.

Running the executable file hello.out [generated by compiling and linking the source in hello.c using the full suite of gcc flags] should generate the following output:

```
$./hello.out
Hello World
$
```

You can generate the executable hello.out in a single step by asking the compiler toolchain to execute the preprocessor, compiler proper, assembler, and linker programs sequentially like this:

```
gcc -std=c11 -pedantic-errors -Wall -Wextra -Werror hello.c -o hello.out
```

Compiling and linking with multiple source files

To review the concepts of function declarations and definitions that were covered in Lecture 4 and to gain experience with writing C programs containing multiple source files, we'll split the code in source file hello.c into multiple source files. Go to slide 35 of the deck and edit C source file hello-defn.c. Understand the purpose of each line of the source file. Compile the source file to an object file exactly as described at the bottom of slide 35.

Go to slide 34 and edit header file hello-decl.h exactly as shown in the slide. This file contains the declaration or prototype of function `hello` [which you've previously defined in source file hello-defn.c]:

```
void hello(void);
```

When header file hello-decl.h is included by any other C source file, the function prototype is inserted into that source file. When the source file is compiled, the inserted declaration will introduce to the compiler the following information: the function name `hello`, the number of inputs and the type of each input [in this case, the function takes no inputs and this is indicated by keyword `void`], and the type of value returned by the function. This function prototype will allow the compiler to determine whether the use of name `hello` in a source file [that includes header file hello-decl.h] is semantically correct. If the function's prototype is not encountered by the compiler when compiling the source file, the compiler toolchain will flag an error message and terminate the compilation process.

Next, go to slide 36 of the deck and edit C source file driver.c exactly as shown in the slide. Let's understand the purpose of each line of this source file. The line

```
#include "hello-decl.h"
```

will make the preprocessor include the contents of file hello-decl.h into source file driver.c and therefore introduce the name `hello` as a function that takes nothing and returns nothing. Notice that the header file is delimited using double quotes rather than angular brackets. The double quote delimiters instruct the preprocessor that it need not search the installation directories of gcc to locate header file hello-decl.h but to instead search the current directory [where the gcc

command is being executed]. Compile source file driver.c to an object file exactly as described at the bottom of slide 36.

Create an executable using the gcc command listed at the bottom of slide 38. Running the executable hello.out should print the following text to the console:

```
$./hello.out
Hello World!!!
$
```

Problem Statement

The submission will replicate the steps from the previous section in a different context. Download files calc.h, calc.c, and calc-driver.c from the lab web page. Your task is two-fold: to provide declarations of *three functions* documented in calc.h and to define these functions in calc.c.

Testing

Begin by ensuring your definitions in calc.c successfully compile with the full suite of gcc flags:

```
gcc -std=c11 -pedantic-errors -Wall -Wextra -Werror -c calc.c -o calc.o
```

You will not be editing calc-driver.c. Instead, you'll be using the calls from function `main` [defined in calc-driver.c] to test your definitions [in source file calc.c]. Separately compiling source file calc-driver.c into an object file calc-driver.o:

```
gcc -std=c11 -pedantic-errors -Wall -Wextra -Werror -c calc-driver.c -o calc-
driver.o
```

After successfully compiling the two source files, the next step is to link the two object files calc.o and calc-driver.o into an executable calc.out:

```
gcc calc-driver.o calc.o -o calc.out
```

Run the executable and redirect the program's output to a file, say your-output.txt:

```
./calc.out > your-output.txt
```

Download output file good.txt containing the correct output generated by function `main`. If your definitions of functions in calc.c are correct, your output in your-output.txt should match the output in good.txt. Rather than manually comparing the contents of two files, you can use command diff in the Linux bash shell to compare your implementation's output with the correct output provided to you, like this:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-output.txt
good.txt
```

Options -y, --strip-trailing-cr, and --suppress-common-lines are described here. If diff completes the comparison without generating any output, then the contents of the two files are an exact match - that is, your source file generates the exact required output. If diff reports differences, you must go back and amend calc.c to remove these differences.

File-level documentation

Every source and header file you submit *must* contain a *file-level* documentation block at the top of the file. The purpose of this documentation block is to identify the author(s) of the source file and to provide other programmers useful information about the purpose of this source file at some later point of time [could be weeks later or months later or even years later]. Here is a *template* of a file-level documentation block:

```
/*!
@file     @todo what is name of this source file?
@author   @todo provide your name & DP login: Nicolas Pepe (nicolas.pepe)
@course   @todo which course is this source file meant for?
@section  @todo which section of this course are you enrolled in?
@tutorial @todo provide Tutorial #
@date     @todo provide date on which you created the file
@brief    @todo provide a brief explanation about what this source file
          does like the example description below:
          This file contains a collection of functions that puts a salmon
          on a cedar plank and smokes the fish for three hours. Remove the
          plank with the fish, throw away the fish, and enjoy the plank.
*//*_____________________________________________________________________*/
```

Make modifications to the template by replacing `@todo` with your information. Providing a file documentation header is mandatory for any submissions you make in this course.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit files calc.h and calc.c.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of gcc options.
   - F grade if your submission doesn't link to create an executable.
   - F grade if the program's output doesn't match the correct output.
   - $A+$ grade if the submission's output matches the correct output.
   - A deduction of one letter grade for each missing file header documentation block. Every submitted file must have one file-level documentation block. A teaching assistant may physically read submitted files to ensure that these documentation blocks are authored correctly. Each missing block may result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade may be later reduced from $A+$ to $B+$. Another example: if the automatic grader gave your submission a C grade and two documentation blocks are missing, your grade may be later reduced from C to F.