CPE-2600 FALL 2024

Systems Programming

Lab 2 Building Programs

Introduction

The purpose of this assignment is to explore how programs are built in the C programming language. You will be guided through how to use the GNU C Compiler (gcc) as well as write several programs and see what steps are taken to translate them from source into an executable.

You may work with other(s) during the lab period but each student should complete all steps and prepare his/her own submission. The goal is to complete this assignment during the lab period, and it is due by the end of the lab period. If you cannot complete the activity by the end of the period and are still present, you may be granted an extension.

Throughout the exercise, collect screen shots and and answer the questions posed [to a Word document]. Be sure to include the questions you are answering along with your answers (e.g. copy question from exercise and add your answer). You will need to submit this to earn credit for the exercise.

Objectives

By the end of this assignment you will be able to:

- Create C source files
- Link C source files together through header files and `include` statements
- Use preprocessor directives
- Display and analyze the output at each step in the compilation of a C program

Background and References

In order to execute a program written in the C programming language, it must first be 'compiled'. To compile something means to "to translate instructions from one computer language into another so that a particular computer can understand them" ~ https://www.oxfordlearnersdictionaries.com/us/definition/english/compile. In our case, that means translating the C source code into instructions that can be executed by our CPU.

The compiler that we will be using is called `gcc`, which stands for the **G**NU **C** **c**ompiler. GNU is a recursive acronym which stands for **G**NU is **n**ot **u**nix, chosen because the behavior of GNU tools are 'Unix like' in the sense that they follow the design of Unix but are free and open source and contain no Unix source ~ https://www.gnu.org/.

Phases of Compilation

In this section you will be analyzing the output at each phase of compilations.

The GNU C compiler utilizes four (4) phases of compilation:

1. Preprocessing
2. Compiling
3. Assembling
4. Linking

At each phase of compilation, gcc performs a portion of compilation tasks, the output of each phase is 'fed in' as input to the next phase, with the final phase outputting the program executable.

For this activity start with the source for hello world:

```c
/*********************************
* hello.c
*
* A hello world program in c
*********************************/
#include <stdio.h>

int main (int argc, char* argv[])
{
     // Print a message to the user
     printf("Hello World!\n");

     // Return to exit the program
     return 0;
}
```

Preprocessing

Including Files

In this phase, gcc takes the C source input and 'pre-processes' it. The input is C source code and the output is also C source code. The difference in the code after preprocessing is that all preprocessor 'directives' have been handled and the resulting source is ready to be compiled. More details on the gcc preprocessor (cpp) can be found here: https://gcc.gnu.org/onlinedocs/cpp/.

SUBMISSION REQUIREMENT: Research the gcc preprocessor.

1. Determine what command 'flag' parameter that needs to be specified to force gcc to stop after the preprocessor phase.
2. What is the purpose of the `#include` preprocessor directive? How does output of the preprocessor differ from your input source?

3. Write the ***exact*** command necessary run the preprocessor on `hello.c` and output the result to `hello.i`. Include the screenshot of a portion of this file in your submission.

   ⓘ There are a couple of ways to accomplish this. Perhaps refer to last week's exercise. The behavior of one of the ways of doing this is a bit different than the -c or -S options.

Macros

One useful preprocessor directive is called a macro. This allows you to define a constant value or operation that is in multiple places in the code. When referenced, the preprocessor does a direct text replacement of the name with what it is defined as. For example, you want to define a constant for the length of an array to be used later. Using a preprocessor macro for this would look like:

> ⓘ We have not yet talked about arrays - we will. Simple enough… Similar to Java, but, you rarely used native arrays in Java (for good reason).

```
// Exploring #define
#define ARRAY_LENGTH 10

int main(int argc, char* argv[])
{
    int array1[ARRAY_LENGTH];
    int array2[ARRAY_LENGTH];

    for(int i = 0; i < ARRAY_LENGTH; i++)
    {
        // Do something cool with array1 and array2
    }
}
```

If you need to change the length of the arrays, you just need to change the value within the `#define` and rebuild your program.

SUBMISSION REQUIREMENT: Research the gcc preprocessor macros.

1. Run gcc on this source and force it to stop at the preprocessor output phase. What do you observe about the output?
   - Save your output and include it with your submission.
2. Can you foresee any issues with this mechanism? As an experiment, change ARRAY_LENGTH from ***10*** to ***"xyz"*** (include quotes). Compile and look at the error message(s). Are the error messages clear?

Compiling

In this phase, gcc takes the pre-processed C source and translates it into assembly language for the target CPU architecture.

SUBMISSION REQUIREMENT: Research the gcc compiler.

1. From your research, determine what command 'flag' parameter that needs to be specified to force gcc to stop after the compiler phase.
2. Write the ***exact*** command necessary run the compiler on `hello.c` and output the assembly language to `hello.s`. Include a snip this file in your submission.

Assembling

While the assembly language output from the compilation phase is written in CPU instructions, it remains in 'plain text' and can not be interpreted directly by the CPU. In the assembly phase, gcc takes the assembly language and 'assembles' it into the binary representation that can be executed by the CPU. More details on the gcc assembler (as) can be found here: https://sourceware.org/binutils/docs-2.23.1/as/index.html.

The output of the assembly phase is call an "object" file. The object file contains executable "machine" code, but by itself is not yet an executable program. There is one more phase in the build process.

SUBMISSION REQUIREMENT: Research the gcc assembler.

1. Determine what command 'flag' parameter that needs to be specified to force gcc to stop after the assembler phase.
2. Write the ***exact*** command necessary run the gcc on `hello.c` and output the object file to `hello.o`. Include a snip this file in your submission.

Linking

This is the final phase of of a build. The output from this phase is a working executable that can be executed directly from the Linux command line. More details on the gcc linker (ld) can be found here: https://sourceware.org/binutils/docs-2.23.1/ld/index.html.

SUBMISSION REQUIREMENT: Research the gcc linker. From your research:

1. What is static linking vs dynamic linking?
2. Determine what 'flag' parameter needs to be specified to force gcc to perform static linking.
3. Write the ***exact*** command necessary and run the compiler on `hello.c` (you may wish to explicitly set the output of gcc to something other than a.out).
4. Compare the file sizes of the statically linked executable to a dynamically linked executable. Include a screenshot showing the relative sizes of both executables. Why are they different?

Building Programs from Multiple Files

While putting all our code in one single source file would work, doing so does not promote good code organization and code reuse. As such, it is often useful to create a program from multiple files.

SUBMISSION REQUIREMENT: Create a program for calculating mathematical operations. This program will be in two (2) source files.

1. `mymath.c` - will contain the implementation of two functions for math operations:
   - `int maxvalue(int bits)` - returns the maximum value that can be represented by `bits` bits.
   - `int numbits(int value)` - returns how many (binary) bits it would take to represent the value of `value`. For this exercise you may assume that `value` is positive and it will be represented with an unsigned binary number. No error checking is needed for this exercise.
   - ⓘ The C standard math library can help out with these two functions - you must use them! Recall the various ways to find library functions. Also, the math functions in the library that you might be interested in will likely accept and return type `double` which we have not yet talked about. So, there may be some conversion between int and double that must occur to match the prototypes given for these functions. Be sure to comment on any issues you had with this.

   ⓘ Do not overthink these functions - they each can be implemented with a single line of code. 2. `main.c` - contains the main function. Your main should exercise the functions in `mymath.c` with a variety of arguments. At a minimum, you should display the maxvalue of 8, 16, and 32 bits, and display numbits for 7, 8, 15, and 16. Print the results of each call to the console. In your `main.c` and/or `mymath.c` experiment with using preprocessor macros to define constant values.

   ⓘ Make sure all code fits the style guidelines and contains all necessary comments and comment blocks.

SUBMISSION REQUIREMENT: Research the gcc compiler:

1. From what you have learned in class and through your research how do you build a program from multiple files?
2. What additional file is needed in order to avoid warnings or errors?
3. What additional argument is needed for gcc to properly build the project? Why? Where did you find this documented?

Create all the necessary files along with the ***exact*** command you needed to run to build your math program. Be sure (out of habit) to inclide the command line option for "all" warnings even though it is not explicitly necessary. Include screen caps of all of your files as well as the output of your program with your submission.

Deliverables

As noted at the beginning of this document, collect various requested screenshots and commentary [in a Word document]. Submit a pdf printout of that document. Be sure to include your name and date at the top. Submit to the Week 2 Lab Assignment in Canvas. It is intended that you be able to submit this by the end of the lab period.