Lab 07: The Compilation Process

Objectives:

- Understand the C compilation process.
- Learn to use Command Line Interface (CLI) of GNU C Compiler (GCC).
- Learn to compile single code files using GCC, CLI
- Learn to compile multiple code files using GCC, CLI

Pre-Lab Reading 1: GNU C Compiler (GCC) Compilation Process

Compiling a C program is a multi-stage process. At an overview level, the process can be split into four separate stages:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

We will be using the following piece of code to illustrate this process. Save this code in a file (using any text editor) and name it **"test1.c"**

```c
#include<stdio.h>

int main(void)
{
    int x = 0;    // A variable to hold some data
    int y = 5;    // and another variable

    printf("The sum of x and y is: %d", (x+y));

    return(0);
}
```

Code Listing 1: A Simple C Program

Pre-processing:

This is the first phase through which source code is passed. This phase includes:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.

In this stage, lines starting with a **'#'** character are interpreted by the preprocessor as preprocessor commands. These commands form a simple macro language with its own syntax and semantics. This

language is used to reduce repetition in source code by providing functionality to inline files, define macros, and to conditionally omit code.

Before interpreting commands, the preprocessor does some initial processing. This includes joining continued lines (lines ending with a \) and stripping comments.

To print the result of the preprocessing stage, pass the **-E** option to gcc:

```
gcc -E “Path\test1.c” -o “Path\test1.i”
```

or call the sub-process **cpp** as follows

```
cpp “Path\test1.c” -o “Path\test1.i”
```

Given the “test1.c” example above, the preprocessor will produce the contents of the stdio.h header file joined with the contents of the test1.c file, stripped free from its leading comments.

```
[lines omitted for brevity]

# 625 "C:/Program Files/CodeBlocks/MinGW/include/stdio.h" 3
 FILE* __attribute__((__cdecl__)) __attribute__ ((__nothrow__)) wpopen (const
wchar_t*, const wchar_t*);

# 3 "Z:\\Programing Fundamentals\\Labs\\Lab 07 Compilation Process\\Test
Code\\test1.c"
int main(void)
{
 int x = 0;
 int y = 5;

 printf("The sum of x and y is: %d", (x+y));

 return(0);
}
```

Figure 1 Output of the Preprocessing Step

Compilation:

The second stage of compilation is confusingly enough called compilation. In this stage, the preprocessed code is translated to assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

The existence of this step allows for C code to contain inline assembly instructions and for different assemblers to be used.

Some compilers also support the use of an integrated assembler, in which the compilation stage generates machine code directly, avoiding the overhead of generating the intermediate assembly instructions and invoking the assembler.

To save the result of the compilation stage, pass the **-S** option to gcc:

```
gcc -S “Path\test1.i” -o “Path\test1.s”
```

This will create a file named **‘test1.s’**, containing the generated assembly instructions. On Windows 7 the following output is generated:

```
    .file   "test1.c"
    .def    ___main;    .scl    2;  .type   32; .endef
    .section .rdata,"dr"
LC0:
    .ascii "The sum of x and y is: %d\0"
    .text
    .globl  _main
    .def    _main;  .scl    2;  .type   32; .endef
_main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    call    ___main
    movl    $0, 28(%esp)
    movl    $5, 24(%esp)
    movl    28(%esp), %edx
    movl    24(%esp), %eax
    addl    %edx, %eax
    movl    %eax, 4(%esp)
    movl    $LC0, (%esp)
    call    _printf
    movl    $0, %eax
    leave
    ret
    .ident  "GCC: (tdm-1) 5.1.0"
    .def    _printf;    .scl    2;  .type   32; .endef
```

Figure 2 Generated Assembly File **"test1.s"**

Assembly:

During this stage, an assembler is used to translate the assembly instructions to object code. The output consists of actual instructions to be run by the target processor.

To save the result of the assembly stage, pass the -**c** option to gcc:

gcc -c "Path\test1.c" -o "Path\test1.o"

or invoke the Assembler as follows:

as "Path\test1.s" -o "Path\test1.o"

Running the above command will create a file named **'test1.o'**, containing the object code of the program. The contents of this file are in a binary format and can be inspected using Notepad++:

Linking:

The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other ones. It will also add pieces containing the instructions for library functions used by

the program. In the case of the "test1.c" program, the linker will add the object code for the ***printf*** function.

The result of this stage is the final executable program. When run without options, gcc will name this file test1.exe (on a Windows machine). To name the file something else, pass the -o option to gcc:

```
gcc "Path\test1.c" -o "Path\test1.exe"
```

Or invoke the linker directly passing the Object file as input:

```
ld "Path\test1.o" -o "Path\test1.exe ...libraries..."
```
(This will only work if the correct path to libraries is provided. Out of scope for this lab.)

This process can be summarized by the following figure:

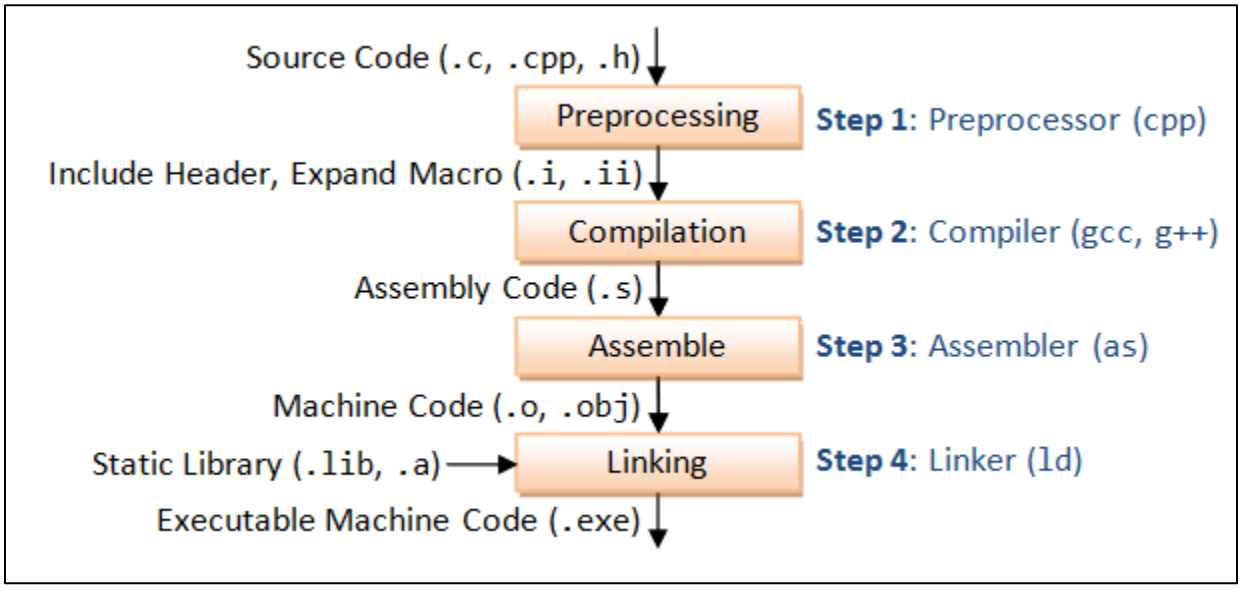

Figure 3 The Four-Step Compilation Process

Pre-Lab Reading 2: Splitting Your Code in Multiple Files

The programs we have seen so far have all been stored in a single source file. As your programs become larger, and as you start to deal with other people's code (e.g. other C libraries) you will have to deal with code that resides in multiple files. Indeed you may build up your own library of C functions and data structures, that you can re-use in your own scientific programming and data analysis. Here we will see how to place C functions and data structures in their own file(s) and how to incorporate them into a new program.

We saw when discussing functions, that one way of including custom-written functions in your C code, is to simply place them in your main source file, above the declaration of the main() function. A better way to re-use functions that you commonly incorporate into your C programs is to place them in their own file, and to include a statement above main() to include that file. When compiled, it's just like copying and pasting the code above main(), but for the purpose of editing and writing your code, this allows you to keep things in separate files. It also means that if you ever decide to change one of those re-usable functions (for example if you find and fix an error) that you only have to change it in one place, and you don't have to go searching through all of your programs and change each one.

Header files

A common convention in C programs is to write a header file (with .h suffix) for each source file (.c suffix) that you link to your main source code. The logic is that the .c source file contains all of the code and the header file contains the function prototypes, that is, just a declaration of which functions can be found in the source file.

This is done for libraries that are provided by others, sometimes only as compiled binary "blobs" (i.e. you can't look at the source code). Pairing them with plain-text header files allows you see what functions are defined, and what arguments they take (and return).

Example:

Let's say you want to write a program that takes one integer input from the user, and determines whether that integer is a prime number or not. Now let's say that you don't want to write your own code for determining primality, so you ask your friend, who you know has written such a function already. He sends you a pair of files (primes.h and primes.c). His header file (primes.h) looks like this:

```
int isPrime(int n); // returns 0 if n is not prime, 1 if n is prime
```

Code Listing 2 The Header File for Prime Numbers **"primes.h"**

So we know a couple of things from this header file. It declares a function prototype for isPrime(). We can see this function takes a single integer as an input argument, and returns an integer value: 1 if the input value is a prime number, and 0 if it is not. Now we know all we need to know in order to use this function (without even looking at the function's source code, which resides in primes.c).

Here is what the primes.c file look like:

```
int isPrime(int n)
{

  // returns 0 if not prime, 1 if prime

  if (n<2) return 0;          // first prime number is 2
  if (n==2) return 1;         // ensure 2 is identified as a prime
  if ((n % 2)==0) return 0;   // all even numbers above 2 are not prime

  int i;
  for (i=3; i*i < n; i++) {   // test divisibility up to sqrt(n)
    if ((n % i) == 0) {
      return 0;
    }
  }
  return 1;
}
```

Code Listing 3 Source File **"primes.c"**

Here is what our program go.c looks like

```
/** go.c
Asks the user for an input and then tells whether the entered number is prime
or not.
   Compile with: gcc -o go.exe go.c primes.c
**/

#include <stdio.h>
#include <stdlib.h>
#include "primes.h"

int main(void)
{
  int N;
  int prime;

  while(1)
  {
    printf("Enter a number");
    scanf("%d", &N);

        if(isPrime(N) != 0)
            printf("The number %d is prime ",N);
        else
        {
            printf("The number %d is not prime ",N);
            break;
        }
  }
    return 0;

}
```

Code Listing 4 The Main Program "go.c"

Search path

The compiler will look in several places for header files that you include with the #include directive, depending on how you use it. If you use include with the angled brackets (e.g. #include <stdio.h>) then the compiler will look in a series of "default" system-wide locations (see Search Path for details). If you use double-quotes (e.g. #include "neuron.h") then the compiler will look in the directory containing the current file. It's possible to add other directories to the search path by using the -Idir compiler option, where dir is the other directory. You might have to do this if you link your code to an external C library that is not part of the standard C library, and does not reside in the usual "system" default locations.

In-Lab Task 1:

Using the Notepad (the Windows text editor), save the program given in Code Listing 1 as a C file (test1.c). Then use the GNU CLI to compile the program and generate the intermediate files like, ***test1.i***, ***test1.s***, ***test1.o*** and finally ***test1.exe***.

You will find the GNU Compiler in C:\Program Files\CodeBlocks\MinGW\Bin.

Hint: You will have to provide the complete path to source and destination files in double quotes (e.g. "Path\test1.c").

In-Lab Task-2:

Using the Notepad (the Windows text editor), save the programs given in Code Listing 2, 3 and 4 as c/h files (primes.h, primes.c, and go.c). Then use the GNU CLI to compile the program and generate the intermediate files like, ***go.i***, ***go.s***, ***go.o*** and finally ***go.exe***.

Post-Lab Task:

Submit a hand written report on what difficulties you faced in the lab, what sources of information did you refer to to solve your problems, and what were the solutions you implemented.

Reference Links:

1. https://www.gribblelab.org/CBootCamp/12_Compiling_linking_Makefile_header_files.html
2. https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
3. https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/compile.html#compilation_steps
4. https://www.gribblelab.org/CBootCamp/
5. https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/
6. https://www.youtube.com/watch?v=VDslRumKvRA
7. https://www.youtube.com/watch?v=N2y6csonII4