Reading Material
Lab Exercise 2: Compiling and Running C

This material introduces you to the ideas behind compiling and running C programs. You can skip this part if you've done it before. **You may be asked questions during marking about material in this document**.

Compiling and Running C Programs

Copy the starting files from */opt/info/courses/COMP26120/problems/ex2* into your *COMP26120/ex2* directory. These files are: *HelloWorld.c, SalaryAnalysis.c, SalaryAnalysis.java, salary-data.txt, makefile*

This is only for you to try out and read before the lab begins.

1.1 Environment

This guide is for compiling and running C programs on a school PC. The compiler we are using is GCC (Type *gcc –version* in terminal to check the installed version).

You are free to work with any IDE in your own system(e.g. *Visual Studio* for Windows, *Xcode* for macOS) as long as the final files work on a lab computer - check it! But support from TA is not given for any problems encountered using an IDE.

1.2 Hello World

Compile *HelloWorld.c* using the command:

```
make HelloWorld
```

You should see a single line of output:

```
gcc -g -std=c99 -Wall HelloWorld.c -o HelloWorld
```

Now run the resulting program using the command:

```
HelloWorld
```

(or, depending on your $PATH variable, you may need to use ./HelloWorld) Again, you should see a single line of output:

```
Hello world!
```

You just used the *make* command, which gets information from your *makefile*, to compile a C program. *make* used *gcc* (the Gnu C Compiler) to create a binary program *HelloWorld*, which can then be run directly without the use of an interpreter (i.e. unlike Java programs, you don't have to use e.g. *java HelloWorld* to run your compiled program).

Here is some basic information about the compilation process. **We strongly suggest you read this.** We have repeated some of the contents here but not all of it (that would have been a waste of our time) and one thing we want you to do in this course is engage with the wider material available.

Did you read that link? Okay, carry on.

1.2.1 Different types of files

Usually compiling C programs involve the following files:

1. **Source code files.** A source code file is a file with extension .c that contains function definitions.

2. **Header files.** A header file is a file with extension .h that contains function declarations and preprocessor statements (see below). A compiler would automatically insert the content of a header file into your program if you add it by using the preprocessing directive #include at the beginning of your source code file. Through this, you are allowed to access externally-defined functions.

3. **Object files.** These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in ".o" on Unix operating systems.

4. **Binary executables.** These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in ".exe" on Windows.

There are other kinds of files as well, notably libraries (".a" files) and shared libraries (".so" files), but you won't normally need to deal with them directly.

1.2.2 The preprocessor

Before the C compiler starts compiling a source code file, the file is processed by a preprocessor. This is a separate program, but it is invoked automatically by the compiler before compilation proper begins.

Preprocessor commands start with the pound sign ("#"). There are several preprocessor commands; two of the most important are:

1. #define. This is mainly used to define constants. For instance,

```
#define BIGNUM 1000000
```

   specifies that wherever the character string BIGNUM is found in the rest of the program, 1000000 should be substituted for it. For instance, the statement:

```
int a = BIGNUM;
```

   becomes

```
int a = 1000000;
```

#define is used in this way so as to avoid having to explicitly write out some constant value in many different places in a source code file. This is important in case you need to change the constant value later on; it's much less bug-prone to change it once, in the #define, than to have to change it in multiple places scattered all over the code.

2. #include. This is used to access function definitions defined outside of a source code file. For instance:

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

causes the preprocessor to paste the contents of <stdio.h> into the source code file at the location of the #include statement before it gets compiled. In this case, we use #include in order to be able to use functions such as printf and scanf, whose declarations are located in the file stdio.h. C compilers do not allow you to use a function unless it has previously been declared or defined in that file; #include statements are thus the way to re-use previously-written code in your C programs.

1.2.3 Making the object file: the compiler

After the C preprocessor has included all the header files and expanded out all the preprocessor commands that may be in the original file, the compiler can compile the program. It does this by turning the C source code into an object code file, which is a file ending in ".o" which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were #included into the file (this is not the same as including the declarations, which is what #include does). This is the job of the linker (see the next section).

In general, the compiler is invoked as follows:

```
gcc -c foo.c
```

This tells the compiler to run the preprocessor on the file foo.c and then compile it into the object code file foo.o. The -c option means to compile the source code file into an object file but not to invoke the linker. If your entire program is in one source code file, you can instead do this:

```
gcc foo.c -o foo
```

This tells the compiler to run the preprocessor on foo.c, compile it and then link it to create an executable called foo. The -o option states that the next word on the line is the name of the binary executable file (program). If you don't specify the -o, i.e. if you just type gcc foo.c, the executable will be named a.out for silly historical reasons.

1.2.4 Putting it all together: the linker

The job of the linker is to link together a bunch of object files (.o files) into a binary executable. This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into library files.

Like the preprocessor, the linker is a separate program called *ld*. Also like the preprocessor, the linker is invoked automatically for you when you use the compiler.

1.2.5 GCC command parameters

You could use the *man* command (e.g. type *man gcc*) to find out more about the command parameters.

- **-g:** The '-g' switch in GCC turns on debugging information in the preferred format for the target.
- **-std=*standard*:** Specify the standard to which the code should conform. "c99" is the revised ISO C standard, published in December 1999.
- **-Wall:** This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning).
- **-o *file*:** Write output to *file*.

1.3 Salary Analysis

Compile *SalaryAnalysis.java* using *javac* and run it with the command:

```
java SalaryAnalysis salary-data.txt
```

to see the output you should expect from the C version of this program.

Compile *SalaryAnalysis.c* using *make SalaryAnalysis*. You should see an error message from *ld* saying:

```
undefined reference to 'lround'
```

gcc has compiled the program without problems, but then called on the linker, ld, to add pre-compiled code from the libraries to your compiled program, and ld cannot find the library code for lround. Look at the synopsis section from:

man lround

The first line says what #include to use (check this in the C code), and the last line says what compiler and linker flags to use - you need to have a linker flag of -lm
To get this, edit the makefile to remove the # (the comment symbol) from the start of the line:

```
LDFLAGS=-lm
```

and use make again to recompile the program. Now run it using:

```
SalaryAnalysis salary-data.txt
```

and compare the output with that from the Java version. **Think about it; what differences are there?**

References

- Caltech CMS: C track: compiling C programs.
- The man page for gcc.