COMP 235 Lab #1: "Hello, World", modules, and *make*

The objective of this lab is for you to practice compiling and linking a multi-module C program in a UNIX environment.

Step 1: "Hello, World" in C.

Use ssh to login to deepblue. Create a new directory for this lab. Using any text editor (pico, vi, emacs, …) that you are familiar with, create a file called main.c that contains the famous "Hello, world" in C.

```
#include <stdio.h>

void main()
   {
     printf("Hello, world!\n");
   }
```

The C compiler is invoked with the command **`gcc`** (guess what it stands for). You use **`gcc`** to:

- compile and link complete programs, in which case the output is an executable file;
- compile modules (see below), in which case the output is an object (*.o*) file;
- link one or more object files into an executable.

Which of these actions the compiler takes depends on the command-line arguments that you provide. Simply invoking "**`gcc`** main.c" will produce an executable called a.out in the current directory. If you want your program to have a different name, say "hello", then use "**`gcc`** main.c –o hello". The "-o" is an example of what is known as a command line option.

Run your program.

Step 2: "Hello, World" with Modules.

The word *module* is overloaded with meaning. In this course, a module is a separate compilation unit developed by a single programmer. In C programming, a module is comprised of a collection of functions. The function prototypes are declared in a *.h* file (a "header" file), and the functions are implemented in a corresponding *.c* file. The *.c* file is compiled into a *.o* file, to be later linked with other *.o* files to create a program. You can think of a module as something equivalent to a Java class file, except the methods are functions and they don't belong to any class.

Create a hello module that implements the following functions:

```
void helloFunc();  // print the string "Hello, world"
void goodbyeFunc();  // print the string "Goodbye, world"
```

You'll need to create two files: hello.h, and hello.c. The header file (hello.h) will look exactly like the function declarations given above. Declarations of functions without implementations are called *prototypes*. Your implementation (hello.c) will look like this:

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

void helloFunc()
{
     printf("Hello, world!\n");
}

void goodbyeFunc()
{
     printf("Goodbye, world!\n");
}
```

Try to compile your module with "**gcc** hello.c". What goes wrong? The compiler tries to create a program, which means it needs a main() routine at which to begin execution. To compile your module, you need to use the –c option: "**gcc** –c hello.c". This tells the compiler not to try and create an executable. Instead, it will create the hello.o object file.

Now, rewrite your main.c to make use of your hello module. It should call the function "helloFunc()" instead of printf. You might naively try the following:

```
void main()
   {
     helloFunc();
   }
```

Trying to compile this version of main.c as before (i.e., "**gcc** main.c") results in two problems. First, the compiler does not know whether you are calling helloFunc() correctly or not; i.e., is that the correct number of arguments (none), and return type (which we're ignoring)? Second, the link stage will reveal that there is no implementation of helloFunc() available.

We get around these problems in the following way. First, in order to make use of the hello module, our main.c needs to `#include` the header file that contains the function

prototypes. We put the name of the header file in double quotes (instead of angle brackets) to tell the compiler to look for it in the current directory.[1]

```
#include "hello.h"

void main()
   {
     helloFunc();
   }
```

Second, when we compile, we also use the "-c" option: "**gcc** –c main.c" will create a main.o file and stop the compiler from trying to link an executable.

The final step is to link hello.o and main.o together to produce an executable: "**gcc** hello.o main.o –o hello" will do it. We could have combined these last two steps as follows: **gcc** main.c hello.o –o hello.

Run your program.

Step 3: Incremental compilation with `make`.

make is an invaluable utility for making executables comprised of more than one object file. You may have noticed that creating the *hello* executable involved a lot of slightly different calls to the C compiler. This becomes extremely tedious and error prone as the number of modules in your program grows.

make is a program that will automatically decide which modules need to be recompiled whenever you wish to create a new version of the executable. When you run **make**, it takes its input from a file called *Makefile*. You create the *Makefile* yourself; it lists a set of dependencies that comprise your executable. A dependency has the following form.

target: <list of things the target depends on, separated by spaces>
        UNIX command to make *target*

The tab character is used to separate the target from its dependencies, and to begin the next line. The idea is that if any of the dependent files are changed, then the target is out-of-date and needs to be rebuilt. It is rebuilt via the command on the next line. For example, the *Makefile* for Step 2 would contain the following.

```
hello:    hello.o main.o
     gcc -o hello hello.o main.o
```

[1] Header files in angle brackets belong to libraries that are part of the C development environment. For example, *stdio.h* is the standard I/O library. This is where – among others – the prototype for `printf()` is found.

```
main.o:   main.c hello.h
     gcc -c main.c

hello.o:  hello.c hello.h
     gcc -c hello.c
```

Note that main.o depends on both main.c and hello.h. If either of these files changes, main.o needs to be rebuilt.

Now, whenever you make a change to any of the modules in your program and you want to recompile, you simply type the comand **`make`** at the UNIX prompt. The **`make`** utility will invoke all the necessary **`gcc`** commands for you. To try this out, add a call to `goodbyeFunc()` in main.c and use **`make`** to rebuild the executable. Notice that since you have changed main.c, **`make`** will first rebuild main.o and then rebuild *hello* (the executable). It will not, however, rebuild hello.o. **`make`** only does what is necessary.

Step 4: Demonstration

In order to receive marks for this lab, demonstrate the ability to build and run your hello program to the instructor.

□