Compiling and Executing Your Program

In this class, we will be using the GNU compiler *gcc*(1) to compile programs. This handout tells you how to do this on the CSIF systems, and then how to execute the program.

In this handout, what you type on the computer is in **`bold Courier typeface`** and what the computer outputs is in `Roman Courier typeface`. The "`%`" is the CSIF's shell prompt.

Creating Your Program

Our 0-th homework assignment is to write a program that will print "hello, world!", in C. We begin by creating a file — let's call it *hello.c* – that contains the program. We need this to be a text file, because the input to the compiler is text; so, we can't produce a Microsoft Word file and use it. It also has to end in "*.c*". Otherwise, the compiler will not recognize it as a C program file, and it will complain. This type of file is called a "source code file" or "source file".

You can create this file in any text editor you like. On the CSIF, the two text editors used most widely are *vim*(1) (or *vi*(1), a less advanced version of the same editor) and *emacs*(1), which is far more powerful and complex. If you haven't used a text editor on Linux, I recommend you start with *vim*. Jonathan Vronsky has written a very brief guide to *vim*; look in the Handouts area to find it. To see a very good, but much more detailed, tutorial on this editor, use the command `vimtutor`:

```
% vimtutor
```

Type either "**`:q`**" or "**`ZZ`**" to exit this tutorial.

Go ahead and create your program file *hello.c*. Make sure it is on the CSIF system before you go to the next section.

Compiling Your Program

Now it's time to compile your program. We will be using the *gcc*(1) compiler. Type the following to the shell prompt:

```
% gcc -ansi -pedantic -Wall hello.c
```

The options (the words in the command that begin with "`-`") mean:

`-std=c11`: use ANSI standard C (this is the C11 standard)

`-pedantic`: require strict conformance to ANSI standard C. This disallows some extensions.

`-Wall`: warn about questionable constructions. This flags some code that is likely to cause problems.

If your program has no errors, you should get another shell prompt with no lines between it and the one where you typed the above command. If you do get some errors, please fix them before proceeding.

If your program is in more than one file, for example the main routine is in *main.c* and functions are in files *func1.c*, *func2.c*, and *func3.c*, simply list all the source files in the compile command:

```
% gcc -std=c11 -pedantic -Wall main.c func1.c func2.c func3.c
```

The order in which you list the source code files does not matter.

At this point, your directory should contain a file named *a.out*. Look for it by running the command `ls` and checking that the file is one of those listed.

Executing Your Program

If you didn't see any error messages in the previous step, the file *a.out* is the compiled program. Type

```
% a.out
Hello, world!
```

Congratulations! You have just compiled and run your first program on the CSIF systems.

Naming Your program

By default, the compiler puts compiled programs in the file *a.out*. If you want your program to be called something else, you can do this in two ways. First, you can tell the compiler to call the output something other than *a.out* by using the "`-o`" option:

```
% gcc -std=c11 -pedantic -Wall hello.c -o hello
```

This causes the compiled file to be named *hello* rather than *a.out*. You can put the "`-o hello`" anywhere after "`gcc`", but the "`hello`" *must* immediately follow the "`-o`".

The second way is to rename the file *a.out*. Once it is created, you can do this using the *mv*(1) command:

```
% mv a.out hello
```

The difference between these two methods is that the first one leaves any existing *a.out* file alone and unchanged. The second overwrites any existing *a.out* file.

make and Makefiles

Typing the full compile command each time can be annoying. More importantly, when you begin writing more complex programs that occupy more than one file, it can be difficult to remember how to create a compiled program from them. To solve this problem, we use a program called `make`.

`make` executes a series of commands in a file called a *Makefile* (or *makefile*). These commands are selected based on which source files have been modified. For now, as we are only dealing with one source file, our *Makefile* will be very simple.

Store the following in a file named *Makefile*:

```
hello:
	gcc -std=c11 -pedantic -Wall hello.c -o hello
```

The second line *must* begin with a horizontal tab (control-I) — do not put blanks before the `gcc`!

Now delete the file *hello* if it exists:

```
rm hello
```

(if you get an error message saying that file does not exist, ignore it). Then type:

```
% make
gcc -std=c11 -pedantic -Wall hello.c -o hello
```

As before, if your program has no errors, you should get a shell prompt right after the line with the `gcc` command.

Now let's look at a couple of other results. *Without deleting the file hello*, execute the *make* command again:

```
% make
make: 'hello' is up to date.
```

That means you already created *hello*, so it won't be recreated. If you change *hello.c*, you will need to delete *hello* to get *make* to remake it. Later on, we'll show you how to build a *Makefile* that will automatically check whether *hello.c* has been modified since *hello* was made, and if so recompile *hello.c*.

Let's say you forget to put a horizontal tab at the beginning of the second line, and instead put blanks there. Here's what happens:

```
% make
Makefile:2: *** missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

If the first character of the second line is anything other than a horizontal tab, you will get a message similar to this. Just go into your *Makefile* and change the first character to a tab.