CSC 180 lab 2: Compiling and multiple files

Mon Sept 17 or Thurs Sept 20, 2001

This lab discusses some of the important basics to do with compiling and working with C program files. It also introduces the fact that (almost) all C programs in fact use multiple files, and you have to ‘‘link’’ them together.

This lab might take you a bit more than the two-hour lab period. If so, you ought to be able to finish it next week, *so long as you get at least into the middle of step 11 this week*, because next week’s lab ought to take you less than the two hours.

1. Since our hello.c program refers to the standard I/O library function ‘‘printf’’, we should be including the relevant ‘‘extern’’ declaration by including the system file stdio.h. In fact you can get away without this for printf(), but in general, determining precisely under which circumstances you can get away without the proper extern declarations is far more difficult than simply including them.

Edit hello.c (e.g. run ‘‘pico hello.c’’) and make hello.c look like this:

```
#include <stdio.h>

int main()
{
    printf("Hello, world\n");
    return 0;
}
```

2. To be able to run this program, you must compile it to machine language, by typing ‘‘cc hello.c’’. If you do not get any error messages, you will now have a compiled version of this program called ‘‘a.out’’. Type ‘‘ls’’ to check. Run your program.

3. Note that in general in unix, a lack of output from a command is *good*. If something goes wrong, you normally get an error message. Normally nothing goes wrong and you don’t get any response except the next shell prompt. If you mistype a file name, you will get an error message. (For example, type ‘‘mv abc def’’ when you do not have a file named ‘‘abc’’.)

4. To log in to another computer over the network, use the command ‘‘ssh’’. There is a big server machine called ‘‘skule’’. Type ‘‘ssh skule’’ to log in there. Then type ‘‘./a.out’’ to run your program on skule. It doesn’t work! The problem is that skule is a different kind of computer than the one you’re sitting in front of, and each kind of computer has a completely different machine language. The C compiler on each machine produces machine language appropriate for that machine. So type ‘‘cc hello.c’’ *when you’re logged in to skule*, and then you can type ‘‘./a.out’’ successfully on skule.

5. Log out of skule by typing ‘‘logout’’ (one word). You are still logged in to the computer you are sitting in front of.

6. Type ‘‘./a.out’’ and note that this file now doesn’t run on the computer you’re sitting in front of. (You could have saved separate versions for the two platforms in two different files. Since the version of unix running on skule is called ‘‘irix’’, perhaps they could be called ‘‘hello.irix’’ and ‘‘hello.linux’’.)

7. Now let us change the program to print the square root of two instead of ‘‘Hello, world’’. The square-root function is in the ‘‘math library’’, and as such is declared in the system include file ‘‘math.h’’. So your program could look like this:

(continued)

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

int main()
{
    printf("The square root of two is %g\n", sqrt(2));
    return 0;
}
```

Use the “cp” command to copy hello.c to a new file, “sqrt.c”, and edit sqrt.c to look like the above (or similar).

8. If you try to compile this file with the command “cc sqrt.c”, you will get an error message stating that “sqrt” has not been supplied. To link with the sqrt function you need to link with the math library, by specifying −lm. So the command would be “cc sqrt.c −lm”. The printf function is in the “C library”, which is linked with by default. That is, you never actually specify “−lc”, but must specify any further libraries.

Compile and run this program.

Newton’s law says that the force on an object, the mass of that object, and the resulting acceleration of that object caused by that force are related by the equation “f=ma”.

We might write a C program, a session with which looks like this:

```
Please enter mass in kg: 3
Please enter acceleration in m/s**2: 5
The force in newtons is 15
```

In this case, the “3” and “5” are input, and the “15” is output (as well as all of the text).
We might also write a different C program, a session with which looks like this:

```
Please enter force in newtons: 15
Please enter mass in kg: 3
The acceleration in m/s**2 is 5
```

Also entirely conceivable is a third C program, a session with which looks like this:

```
Please enter force in newtons: 15
Please enter acceleration in m/s**2: 5
The mass in kg is 3
```

All of the data values should be represented as double-precision floating-point values.

In this lab, these three C programs will share some code. However, the text strings and the actual calculation involved will differ.

I am supplying you with a file called “ma.c”; you will likely want to re-read this paragraph after you examine that file. “ma.c” supplies a function “calc()” which performs the calculation; it *also* supplies three string variables which contain the appropriate text snippets. The two items for the user to enter are described in the variables name1 and name2, and the result is described in the variable name3. You’ll see a data type “char[]”; this is suitable for strings, as we will discuss later; for now we’ll just take “char[]” as meaning “string”.

This ma.c file can then be attached to a main() function which you will write (in a separate file called main.c) which will use these three strings (with printf formats of %s for string), get the appropriate inputs, and call calc() and display the result. So the main.c file will be independent of which of the three variables are input and output.

(continued)

After you get main.c working with my ma.c, you can then write the files fm.c (in which we input force and mass, and output acceleration) and fa.c (in which we input force and acceleration, and output mass). (These three files are all named after their pair of inputs.) These other C files can then each (one at a time) be compiled with the *same* main program.

The main.c file has to refer to the objects name1, name2, name3, and calc() in the other file. A sample main.c file contains the appropriate ‘‘extern’’ declarations to refer to these objects.

So, your tasks are as follows:

9. Copy my ma.c file to your own directory using the command:

```
cp ~ajr/lab2/ma.c ma.c
```

You will not change this file. If you change it for exploratory purposes, you should copy it again afterwards; if this were an assignment, your main.c would be *required* to work with a verbatim copy of my ma.c.

10. Copy my main.c starter file to your own directory using the command:

```
cp ~ajr/lab2/main.c main.c
```

11. Write the main() function by editing main.c. Remember that on assignments (and in ‘‘real life’’) you will graded on the quality of the program you produce, not only on its output; design your programs to be read by people as well as by computers. Use standard indentation and formatting; include appropriate comments.

Remember that for a `double`, the printf format is %g but the scanf format is %lg.

12. To link together your main.c and the supplied ma.c, you could use the command:

```
cc main.c ma.c
```

Actually you will probably want to call the result ‘‘ma’’ rather than ‘‘a.out’’ (especially since you will shortly have more than one), so you will instead use:

```
cc -o ma main.c ma.c
```

The ‘‘−o’’ stands for ‘‘output’’. I.e. the output of the compilation goes into the file ‘‘ma’’.

13. (a biggie) Write your fm.c and fa.c. Similar compilation commands can produce the other two compiled programs, e.g. ‘‘cc main.c fm.c’’ and ‘‘cc main.c fa.c’’. Make sure that all of your programs work: your same main.c with each of *my* ma.c and *your* fm.c and fa.c.

14. If you type ‘‘fm’’ on the computer known as ‘‘skule’’, what happens? You get an unrelated program called ‘‘fm’’ (which is short for ‘‘file manager’’, in that case). This is the main reason we always run our own programs as ‘‘./a.out’’ (or ‘‘./fm’’, etc). A single period is a name for the current directory.

(Also, ‘‘dot’’ might not be in your ‘‘path’’, the list of directories in which the shell (command interpreter) searches for commands you type such as ‘‘a.out’’. This search procedure is only applied to command names which do not have a slash in them anywhere.)