CS 0449 – Makefiles

Background

Sure… you *could* write your entire program in just a single file. However, that's not a very practical thing to do. Rummaging around something that's even just hundreds of lines of code can be a grueling process you'll never want to repeat.

So, let's learn how to manage a project that is split up across several files. As we have seen in lecture, multiple files can be compiled independently and then merged together in a process called **linking**. Generally, these two phases use different tools behind the scenes. Thankfully, though, the compilers will often hide that process and invoke it for us.

In this worksheet, we will create some utility functions in a separate file and then call them from a main C file. We will learn how to compile each file independently. And then we will learn about **make** which is a tool which helps us manage and simplify the build process. We've already seen and used make on our programming assignments so far! Now, we can claim that power for ourselves.

Procedure

1. Login via SSH to `thoth.cs.pitt.edu`

2. Change directories to one you created for the work for this class.

3. Then, create a directory for this worksheet and navigate to it:

```
mkdir makefiles
cd makefiles
```

4. The first step will be to create our main C file (using the terminal editor of your choice):

```
nano main.c
```

5. Now enter our initial code:

```
int main(void) {
    return 0;
}
```

6. Save the file

7. We now need to compile this. Which is rather straightforward when it is just a single file. However we want to compile and link in separate steps for now:

```
gcc -c main.c
```

8. The "`-c`" argument to gcc will create a `main.o` object file instead of link an entire executable. You can see it if you use the `ls` command to list the directory. We can now link the object file with the C standard library and create an executable called `main` by using the following:

```
gcc -o main main.o
```

9. Now, what if we had another file? Let's create a util.c file that contains some functions we would like to use. Create the following file and fill it with the provided code:

```
nano util.c
```

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }

    return n * factorial(n - 1);
}
```

10. We don't need to do much inside this C file. However, how do we call it from our `main.c` file? Well, it has to know what is defined in `util.c`. So, we have to write a separate file that contains a listing of functions and definitions useful to our second file. So, let's create a `util.h` file:

```
nano util.h
```

11. Place this code inside:

```
int factorial(int n);
```

12. Notice that this "**header file**" only needs the strict definition of any functions you want to expose to your C files. Here, it suggests that you can call a function named `factorial` and it will return an `int` and takes an `int` as input. This is called a **function prototype**. The rest of your code does not need to know the implementation details (such as it being recursive.)

13. Now we can build this file as well: (Notice, we do not do anything with the header file, util.h)

```
gcc -c util.c
```

14. Now let's modify our main.c to have it call the factorial function:

```
nano main.c
```

And let's use the following code to have it call the factorial function:

```c
#include "util.h"

int main(void) {
    printf("%d\n", factorial(5));
    return 0;
}
```

Yes, we `#include` the `util.h`. This effectively just copies-and-pastes the contents of `util.h` into this file. This is why we need the separate files since copying the actual function *duplicates* the code for the factorial function and multiple versions of a function will cause us grief later on.

Could we just type the function declaration contained in `util.h` into the top of `main.c` instead of including the file with the preprocessor? Yes! But it sure is more annoying to change...

15. And we can re-compile our main function:

```
gcc -c main.c
```

16. And then re-link our executable (including the code in util!):

```
gcc -o main main.o util.o
```

NOTE: why did we not have to re-compile `util.c`? It did not change! You only have to recompile files that change, which is the power of linking. We will potentially explore this in detail later in the course. For now, we will exploit this in our tooling with make!

17. However, this gives us an opportunity to do something smarter. A tool called **make** is designed to invoke commands, such as each of the **gcc** commands above. Furthermore, it can be written to *only run commands* on files that have changed. Let's look at the Makefile for our little project so far. make looks for a particular file in the current directory. Create the file we need here (note the capitalization):

```
nano Makefile
```

Write the following and save the file:

```
main: main.o util.o
    gcc -o main main.o util.o

main.o: main.c
    gcc -c main.c

util.o: util.c
    gcc -c util.c
```

NOTE: It is VERY important that the indentation you use are strictly tabs. That is, you must tab over the "gcc -c util.c", etc using your keyboard's tab key. If you use spaces (or your editor replaces tabs with spaces) you will get errors from Make such as:

```
Makefile:2: *** missing separator.  Stop.
```

Back at your shell, run your makefile by typing this command:

```
make
```

Reviewing our work so far…

When written correctly, it will build your executable when you type `make`. The Makefile is a set of rules specified before a colon (`:`) such as “main.o” and “util.o” and “main”. After each rule is a set of files. It will run the commands under the rule if the specified list of files have been changed since the creation of the targetted file.

For instance, main.o is created using main.c, hence the second rule in the file is used. If main.o does not exist, it is created by the given gcc command. If it does exist, then it determines if it should recreate the file by looking at the files listed in the rule. If any of the files are newer than main.o (in this case if main.c has been changed,) it will re-execute the listed commands.

As you can follow, the rules may list files that are the targets of other rules. The executable `main` is defined by the first rule in our Makefile. The main target should be specified first in your Makefile. Here, the `main` executable is the target and it lists `main.o` and `util.o` as its dependencies. Therefore, the other rules must be logically satisfied before main can build.

Makefiles are very useful. They can be written for any task, not just C and not just for programming. You can create tasks for building documentation or testing your program. Invoking a specific task is as simple as specifying it as an argument to make (targets do not have to be real files!):

```
main: main.o util.o
    gcc -o main main.o util.o

main.o: main.c
    gcc -c main.c

util.o: util.c
    gcc -c util.c

clean:
    rm *.o main
```

```
make clean
```

This will reset your environment so that all files must be compiled again. Very useful task you will find in many Makefiles. Consider what you would have to write in your Makefile when adding another file to your project.

Your turn

Add another C file called "`mystuff.c`" or something like that. Add any function you would like! If you are very stumped for an idea, write a basic function that takes two integers and adds them together. Remember to also create a `mystuff.h` that contains a function prototype for your function.

Then, modify your main.c to call that function and use it in some way.

If you build your project at this point, you will get an error about not finding the implementation of the function you've written. You have to compile and pass along the **object file** for that C file you have just created. Now, modify your Makefile to compile and link this new file.

Once that is done, you can "make" your project and then see the results!

Happy coding!

For more information

The documentation for Make including some examples and tutorials can be found here:
https://www.gnu.org/software/make/manual/html_node/Introduction.html