Makefiles

As your programs get larger and larger, you'll A) want to separate your code into different files to make for an easier reading / coding experience, and B) compile all of these files (not just the main). Consider the program `doMath.c` from the previous lesson:

`doMath.c`

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

#define TEST 1

int addition(int firstNum, int secondNum){
  return firstNum + secondNum;
}

void testAddition() {
  int firstNumber = 10;
  int secondNumber = 20;
  int expectedOutput = 30;

  int response = addition(firstNumber, secondNumber);

  if (expectedOutput != response) {
    fprintf(stderr, "testAddition has failed! actual: %d  expected %d", response, expectedOutput);
    exit(1);
  }

  return;
}

void runTests() {
  testAddition();
}

int main(int argc, char** argv){

  if (TEST) {
    runTests();
    printf("ALL TESTS PASS!");
```

```
    return;
  }

  if (argc != 3) {
    fprintf(stderr, "Bad Input!");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);

  int result = addition(num1, num2);

  printf("%d\n", result);
  return EXIT_SUCCESS;
}


```

This program is getting pretty long. What if we broke it up a little bit? First, let's extract our addition function to a new file:

`mathOperations.c`

```
int addition(int firstNum, int secondNum){
  return firstNum + secondNum;
}
```

`doMath.c`

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

#define TEST 1

int addition(int, int);

void testAddition() {
  int firstNumber = 10;
  int secondNumber = 20;
  int expectedOutput = 30;

  int response = addition(firstNumber, secondNumber);

  if (expectedOutput != response) {
```

```
    fprintf(stderr, "testAddition has failed! actual: %d  expected %d",
response, expectedOutput);
    exit(1);
  }

  return;
}

void runTests() {
  testAddition();
}

int main(int argc, char** argv){

  if (TEST) {
    runTests();
    printf("ALL TESTS PASS!");
    return;
  }

  if (argc != 3) {
    fprintf(stderr, "Bad Input!");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);

  int result = addition(num1, num2);

  printf("%d\n", result);
  return EXIT_SUCCESS;
}
```

When we go to compile this file, we then write:

```
gcc mathOperations.c doMath.c -o doMath
```

All in all, this is fine. However, as programs grow in size, the file sizes also grow, let's extract our tests into a separate file too:

`tests.c`

```
void testAddition() {
  int firstNumber = 10;
```

```
  int secondNumber = 20;
  int expectedOutput = 30;

  int response = addition(firstNumber, secondNumber);

  if (expectedOutput != response) {
    fprintf(stderr, "testAddition has failed! actual: %d  expected %d",
response, expectedOutput);
    exit(1);
  }

  return;
}

void runTests() {
  testAddition();
}
```

Now that we've extracted these tests, we still need declarations in our main, so let's add those:

`doMath.c`

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

#define TEST 1

int addition(int, int);
void testAddition();
void runTests();

int main(int argc, char** argv){

  if (TEST) {
    runTests();
    printf("ALL TESTS PASS!");
    return;
  }

  if (argc != 3) {
    fprintf(stderr, "Bad Input!");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);
```

```
25
26    int result = addition(num1, num2);
27
28    printf("%d\n", result);
29    return EXIT_SUCCESS;
30  }
```

Now we've extracted out tests and our functions into other files! The downside though, is that our main file is still a little crowded looking with the function declarations up top. If we had a ton of functions this would take up entire screens worth of real estate! What if we extracted those into header files for each of our files? If we did that, our whole program would look like:

`mathOperations.h`

```
1  int addition(int, int);
```

`mathOperations.c`

```
1  int addition(int firstNum, int secondNum){
2    return firstNum + secondNum;
3  }
```

`tests.h`

```
1  void testAddition();
2  void runTests();
```

`tests.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  void testAddition() {
6    int firstNumber = 10;
7    int secondNumber = 20;
8    int expectedOutput = 30;
9
10   int response = addition(firstNumber, secondNumber);
11
12   if (expectedOutput != response) {
13     fprintf(stderr, "testAddition has failed! actual: %d  expected %d",
   response, expectedOutput);
```

```
    exit(1);
  }

  return;
}

void runTests() {
  testAddition();
}
```

`doMath.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "mathOperations.h"
#include "tests.h"

#define TEST 1

int main(int argc, char** argv){

  if (TEST) {
    runTests();
    printf("ALL TESTS PASS!");
    return;
  }

  if (argc != 3) {
    fprintf(stderr, "Bad Input!");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);

  int result = addition(num1, num2);

  printf("%d\n", result);
  return EXIT_SUCCESS;
}
```

Now we've got our files pretty clean, each is doing their own thing, but compiling this can huge pain:

```
gcc mathOperations.h mathOperations.c tests.h tests.c doMath.c -o doMath
```

The longer your program gets, the more files you'll need to add. In an IDE's environment, you'd only need to press the play button, but via the command line, you have to do a bit more! That's why `make` exists!

`make`

To use make, we'll need to have a special file called the `Makefile`. The Makefile is something that the make utility will use to determine what needs to be recompiled, and what doesn't. And it's a ton easier to type `make` than to type out `gcc mathOperations.h mathOperations.c tests.h tests.c doMath.c -o math`.

How a makefile works is that you first, indicate the dependencies. Because we're making the object file, we'll be using `gcc`'s flag':

```
mathOperations.o: mathOperations.c
  gcc -c mathOperations.c
```

Now if you then type:

```
make
```

You'll see you've created a `mathOperations.o` file in your project!

NOTE: When working with a makefile, make sure that you use a tab for the hanging material! If you use spaces you'll likely see an error that looks like:

> Makefile:<LINE_NUMBER>: *** missing separator. Stop.

You could, if you wanted to, hard code a make file for every project, but often times, most projects follow the same architecture, so you can abstract out much of the heavy lifting by using *macros*:

```
GCC = gcc
CFLAGS = -g -Wall -Wshadow

doMath: mathOperations.o tests.o doMath.o
  $(GCC) $(CFLAGS) mathOperations.o tests.o doMath.o -o doMath

mathOperations.o: mathOperations.c mathOperations.h
  $(GCC) $(CFLAGS) -c mathOperations.c mathOperations.h

tests.o: tests.c tests.h
  $(GCC) $(CFLAGS) -c tests.c tests.h
```

```
12
13 doMath.o: doMath.c
14   $(GCC) $(CFLAGS) -c doMath.c
```

At lines 1 and 2, we're defining our compiler and the flags that we wish to use for everyone, lines 7, 10, and 13 are for each of the files individually, while line `4` is specifically for the program executable `doMath`. If we type in bash:

```
1 make
```

> gcc -g -Wall -Wshadow mathOperations.o tests.o doMath.o -o doMath

We then compile the files and we've done it with one word instead of a ton! Additionally, try typing `make` again:

```
1 make
```

> make: `doMath' is up to date.

Using macros like we did above is great, and it gives a bit more clarity as to what's happening, but we can even further abstract our makefile! We can take our dependencies and our object files, and make those into macros and let `make` do all the work for us!

```
1  CC = gcc
2  CFLAGS = -Wall
3  DEPS = mathOperations.h mathOperations.c tests.h tests.c doMath.c
4  OBJ = mathOperations.o tests.o doMath.o
5
6  %.o: %.c $(DEPS)
7          $(CC) $(CFLAGS) -c -o $@ $<
8
9  doMath: $(OBJ)
10         gcc $(CFLAGS) -o $@ $^
```

What this file does is lets us set all of our dependencies and our object files entirely to to the `DEPS` and `OBJ` variables. We then let make do all of the work for us by using the automatic variables `$@` which uses the filenames of the targets of the rule (our dependencies), and then are named with `$<` which targets the prerequisite.

In doMath we use the `$@` again to target the objects, but then use `$^` to target the names of all the prerequisites!

Now, for every project you use, you only need to change your dependencies and your objects on lines 3 and 4!