When we start dealing with many files that are split into multiple assemblies (the C equivalent of java packages) it becomes difficult to manage the project with gcc. Fortunately there is a utility called make that allows us to automate the build process.

Make uses files called makefiles (literally files named makefile) to organize C projects into assemblies and automate calls to gcc. Let's start with an example.

```
main: module1.o module2.o
        gcc –o program module1.o module2.o
```

That's a lot to consume at once, what's important is to see the common pattern.

```
target: dependencies
        gcc commands
```

The make utility uses a makefile try to "make" a target. A target specifies which files and/or other targets it depends on. Once those dependencies have been resolved the make utility runs the gcc commands listed for that target. Let's walk through the first example together.

Let's say we are trying to make the "main" target. Main depends on the files module1.o and module2.o. make is smart enough to know that it can create those files by running

```
gcc –c module1.c
```

The –c flag tells gcc to create a file called module1.o. This is an assembly.

It then does the same for module2.o.

Now that module1.o and module2.o have been made make will finally run the gcc commands for the main target.

```
gcc –o program module1.o module2.o
```

This will create an executable named program that links the module1.o and module2.o assemblies.

This is effectively just a convoluted way to say

```
gcc -o program module1.c module2.c
```

So why go to the bother?

Well, if we update module2.c without updating module1.c make won't bother to remake module1.o since the file is newer than the module1.c file that it depends on.

While this isn't a large gain for a project that is only two files large it can mean massive gains for a very large project, especially if large parts of the project don't get updated regularly.

Another useful tool when dealing with large projects is the custom header file. In the past we have included the stdio.h and stdlib.h header files. Fortunately we can define our own. Why do I say fortunately? Well, custom header files can be used to ease some of the inconveniences that come with programming in C. Allow me to demonstrate.

```
double squareFunc(double base){
        return powerFunc(base,2);
}
double powerFunc(double base, int exp){
        double product=1;
        int I;
        for(i=0;i<exp;i++){
                product*=base;
        }
        return product;
}
```

These two functions provide the ability to raise any number to any whole power. Unfortunately the above code doesn't compile. Since the squareFunc references the powerFunc before the powerFunc is defined C has to guess at what type it returns. Unfortunately C always guesses int. So the above code won't compile because C assumes powerFunc returns a double, but then it encounters the definition which says that it returns a double.

Of course, we can fix this by simply defining the powerFunc first, but what would we do if we had two files that called each other? What if a file1 function called a file2 function, AND a file2 function called a file1 function? Which file would we compile first in that case?

We can solve this problem by defining custom header files.

A custom header file is any file with the .h extension. Technically speaking header files can contain any c code, but generally we restrict them to declarations only.

A declaration is a statement that tells the compiler to expect a specific thing with a specific type. For example:

int i;

This is a declaration because it tells the compiler to expect and int named i, but since i isn't given a value the compiler doesn't actually ask the operating system for the memory to hold it. When we assign a value it is called a definition.

To solve our problem above (with squareFunc and powerFunc) we can use a special type of declaration called a function prototype (a statement which declares a function). For example:

```
double squareFunc(double base);
double powerFunc(double base, int exp);
```

These statements tell the compiler to expect functions with certain names, input parameter types, and return types, but doesn't actually define them.

If we put them in a header file (say math.h) and then include them in our above file we can tell the compiler what the return type of powerFunc is before we define squareFunc. This way the compiler doesn't have to guess, so our code will compile. Here's what that looks like:

```
#include "math.h"
double squareFunc(double base){
        return powerFunc(base,2);
}
double powerFunc(double base, int exp){
        double product=1;
        int I;
        for(i=0;i<exp;i++){
                product*=base;
        }
        return product;
}
```

Note that we used #include "math.h" instead of #include<math.h>. The " syntax tells the preprocessor to look in our local directory for the file (instead of looking wherever it keeps stdio.h)

We can also include enum, union, and struct declaration in a header file. When we are talking about these constructs the meanings of "declaration" and "definition" get a little hazy. For example:

```
enum myEnum {TYPE1,TYPE2};
```

Though it looks like we're defining the enum, we're really declaring it. At this point we're not telling the compiler to store any new information, we're simply telling it to expect enum myEnums which can have the values TYPE1 or TYPE2. We are only making a definition when we assign a value to a variable, as this is the only time the compiler needs to allocate memory (other than function calls, which are not declarations OR definitions). For example:

```
enum myEnum aVariable = TYPE1;
```

This is a definition since the compiler needs to allocate memory to hold the TYPE1 value in the aVariable variable.