Using Multiple Source Files

This handout is intended primarily for people who intend to be CS majors or who might want to do real programming in the future. When you are working on a large project you will inevitably want to break the program up into multiple source files, that is to say you will want to have multiple files with the .cpp extension that have code in them. You will also likely have multiple header files (.h or .hpp files). When this is the case you have to compile things a bit differently. There are also certain guidelines that most people will follow on how to break up files.

Breaking things up

The selection of how to break things up is quite simple and straightforward in OOP because the classes provide you with an obvious delineation between pieces of code. Because of this it is customary to put different classes in different files. Sometime you might see multiple classes in a single file, but you are also likely to run into people who will seriously frown on this. For each class that you have, you should have both a .cpp and a .h (or .hpp) file. The name of the class is often used as the name of the file. This makes it much easier to find things when the number of files becomes truly large. The class declaration needs to be in the header file but it is wise to make it so that this contains only declarations of methods and not definitions. You will often also have a separate file for the main function though this isn't required. You could just as easily put main in the .cpp file for your primary class in the program.

To see how this might work let's look at the case of a bank program that has two classes, customer and account. The bank has multiple customers and each customer can have several accounts associated with him/her. For example I might have both a savings account and a checking account under my name at the bank. I would probably break up this program into 5 files: Account.h, Account.cpp, Customer.h, Customer.cpp, and Bank.cpp. The two .h files contain only the class declarations for those two classes while the .cpp files contain only the implementations of the functions for those functions (as well as declarations of any static but not const data members). The .cpp files would have #include directives for any of the header files they require. That is, they include the .h files for the classes that they use.

Compiling with Multiple Files

Having include files does not complicate compiling because the #include directives tell the compiler which header files to use in compiling a course file. However, when you have multiple source files you have to do some things a bit differently because you can only tell the compiler to compile a single source file at a time. Before looking at what has to be done we should examine what exactly the compiler does again. While we typically call it a compiler in general what we are referring to actually performs three separate tasks: preprocessing, compiling, and linking. The preprocessor cleans up the text and deals with the lines that start with a # (like the #include statements). The compiler takes the output of the preprocessor, parses it and creates what is called an object file (no relation to what we call objects in OOP). Lastly, the linker takes one or more object files and produces the executable.

We have only dealt with the case where everything was defined in one source file. In this case we execute the command

```
g++ source.cpp
```

where source.cpp is the source file we want to compile. This creates an executable with the default name a.out. We could also specify an alternate name for you output with the –o option:

```
g++ -o assn4 assn4.cpp
```

Then the executable will be called assn4 instead of a.out. This still tries to execute all three steps in the "compile" though. What we need is the ability to force it to skip the link stage at first. We can do this with the –c option. So if we use the above bank example again we would want to execute the following commands to compile to object files all of the source files.

```
g++ -c Account.cpp
g++ -c Customer.cpp
```

```
g++ -c Bank.cpp
```

These create object files with the default names where .cpp is replaced with .o. So we get the files Account.o, Customer.o, and Bank.o. In order to get the executable we need to link all of these files. This is done with a g++ command that specifies all of the object files.

```
g++ Account.o Customer.o Bank.o
```

This again will use the default executable name of a.out. We could again use the –o option to make an executable with the name Bank as follows.

```
g++ -o Bank Account.o Customer.o Bank.o
```

This is a bit of a pain to do every time you want to compile your program. In addition, if a file has not been changed there is typically no need to create a new object file for it. This is the reason the make command exists for Unix (and Linux). make allows to specify what files we want to create and what files they depend upon as well as how to create the file in question. The make command uses specially formatted test files for input called makefiles. By default, make will use the file named "makefile" or "Makefile". While make has lots of power and MANY options we will only look at the most basic usage here. Feel free to do a man on make for more info. O'Reilly also has a book on make if you get really interested. A makefile can contain an arbitrary number of directives. Each directive has the following format:

```
file: dependency1 dependency2 dependence3 …
	commands
```

Here file is the file that the command creates. The dependencies are the files that it uses at input, the ones that is depends on. When make is executed with the given makefile it checks if any of the dependency files have changed since the file was last changed. If one or more have then the commands are executed. The commands should recreate the file. The commands have to be tabbed in, not just spaced in a few times, but tabbed in. I've been told that this makes it so notepad can't be used to edit makefiles. You should just use vi to write and edit them. Given the example above you would have a makefile with the following 4 directives.

```
Bank: Bank.o Account.o Customer.o
	g++ -o Bank Bank.o Account.o Customer.o

Bank.o: Bank.cpp Account.h Customer.h
	g++ -c Bank.cpp

Account.o: Account.cpp Account.h Customer.h
	g++ -c Account.cpp

Customer.o: Customer.cpp Account.h Customer.h
	g++ -c Customer.cpp
```

Note that this assumes complete interdependencies for the include files. You won't have this with a larger project. For example, even here it is possible that Bank.cpp might not actually use Account.h in which case that wouldn't have to be in the list of dependencies.

Submitting Multiple Files

If you have multiple source files to submit for an assignment there are a number of ways that you can get them all too me. Putting them in one big text file is not acceptable. Putting them as multiple attachments will work but is not ideal. Instead, you should use the tar program to group them into a single file and send that as an attachment or as a uuencoded file. The tar command has many options (doesn't everything seem to) but you only need two. The command you will use will be something like this.

```
tar cf assn4.tar assn4.cpp Student.cpp Student.h
```

This creates a file called assn4.tar, which has in it the contents of the other three files listed. You can specify any number of files to go in a tarball and the specification can include wildcards (* and ?). This is when create separate directories for different assignments can really start to come in handy. To send this file as a uuencoded file, you would execute the following command:

```
uuencode assn4.tar assn4.tar | mail mlewis@cs.trinity.edu
```

The uuencode command converts binary data (like a tar file) into a format that can nicely be sent through the mail. The first name is the file you are sending me, the second is the name it will become when I uudecode it. That outputs a bunch of text, which is piped to the mail program. Remember to use my CS account.