Make Instructions

By Sean Dixon, Brady Field, and Paul Irwin

A. Introduction

As programming projects grow larger, it grows exponentially more difficult to manage the individual components. For instance, if a new class is implemented, one could simply leave the definition and implementation of it in the main file; but if this is done many times, the code quickly becomes both unreadable and unmaintainable. This is why separate compilation was developed. Make is a utility that enables easy maintenance and tracking of separate compilation.

B. List of Materials

* Unix-based OS
* Compiler
* Text Editor/IDE
* Basic C++ skills
* Source (.cpp) file with a class

C. Directions

## C.1 - Making the Header file

1. Launch the terminal.
2. Navigate to the directory containing the source file.
3. Compile the program to verify that there are no syntactical errors.
4. Open the source file in the text editor.
5. Locate the class declaration.
6. Highlight and cut all of the text of the class declaration.
7. Open a new file in the same directory.
8. Paste all of the code taken from step 6 into this new file.
9. Save the file with a name like myClassName. The extension of the file MUST be .h! (example: myClassName.h)
10. Make note of the class name. At the top of the file, type

#ifndef MYCLASSNAME\_H

#define MYCLASSNAME\_H

where MYCLASSNAME is the name of the class. The text here MUST be all capitals. The #ifndef is not.

1. At the end of the file, add the line

#endif

This should be the last line of the file.

1. Save both files, then compile the source file. Correct any errors.
2. For an example of what the header file should look like after following the directions in this section see Figure 1.

|  |
| --- |
| timeH.pngFigure 1: time.h  Author: Brady Field |

## C.2 - Making the Implementation File

Navigate to the source file.

Highlight and cut all of the text of the class definition code.

Open a new file in the same directory.

Paste all of the code taken from step 2 into this new file.

Include any libraries that would normally be needed for these class functions to operate correctly. A good rule of thumb is to copy the include statements from the source file.

Add a new line beneath the include statements

#include “myClassName.h”

where “myClassName.h” is the name of the header file created in section C-1.

\*Note that the #include directive is in quotes (“ ”), and not angle braces (< >).

Save the new file with a name like myClassName. The extension of the file MUST be .cpp! (example: myClassName.cpp)

Save what’s left in the original file. This will be our driver file. It should be left with the main program function and any functions that are not a part of the class. Make sure this file also has the include statement #include “myClassName.h”

For an example of what the implementation file should look like after following the directions in this section see Figure 2. For the driver file see Figure 3.

|  |
| --- |
| timeCPP.pngFigure 2: time.cpp  Author: Brady Field |

|  |
| --- |
| driverCPP.pngFigure 3: timeDriver.cpp  Author: Brady Field |

## C.3 - Making the Make file

\*Note - Single line comments are supported in makefiles. The syntax is a pound

sign (#) followed by text (ex. #This is a comment).

# Open a new file in the same directory called “makefile”.

# On the first line, type

myClass.o : myClass.cpp myClass.h

where myClass.cpp is the name of the file created in section C.2. Note the

spaces before and after the declaration. Also note that there are no semicolons.

# On the next line, type

(tab)g++ -c myClass.cpp

The lines from steps 2 and 3 make the first instruction. (See Figure 4)

|  |
| --- |
| timeO.pngFigure 4: time.o  Author: Paul Irwin |

# On a new line, make a new instruction by typing

myMainFile.o : myMainFile.cpp myClass.h

(tab)g++ -c myMainFile.cpp

Make sure that there is nothing but a tab in front of the second line. (See Figure 5)

|  |
| --- |
| driverO.pngFigure 5: timeDriver.o  Author: Paul Irwin |

# On a new line, make a new instruction by typing

a.out : myClass.o myMainFile.o

(tab)g++ myClass.o myMainFile.o

Make sure that there is a tab and nothing else in front of the second line. (Figure 6)

|  |
| --- |
| aOut.pngFigure 6: a.out  Author: Paul Irwin |

# On a new line below all the other, make a new instruction by typing

clean :

(tab)rm a.out myClass.o myMainFile.o

Make sure that there is a tab and nothing else in front of the second line. (Figure 7)

|  |
| --- |
| clean.pngFigure 7: clean  Author: Paul Irwin |

C.4 - Using the Make file

1. In the terminal, type make on the command line. Execute the command.
2. If compiling was successful, make will have created an executable called “a.out” that can be executed from the command line. (Figure 8)

|  |
| --- |
| made.pngFigure 8: successful make  Author: Sean Dixon |

1. Typing the command make clean removes all of the object files generated by the make process in compiling the code. For an example of the files before and after a make and a make clean command see Figure 9.

|  |
| --- |
| makeclean.pngFigure 9: make clean  Author: Sean Dixon |

D. Troubleshooting

Q: How can I check the steps of the makefile?

A: Each instruction created in a makefile execute one at a time. The first make instruction (made in section C-3 above) compiles the .h and .cpp files into .o files. When g++ is called with a .cpp argument, this step is executed anyway, in the background. Make files need to execute these steps one at a time which is why we must create the instructions one at a time. That said, the answer to the above question is that each instruction we created in the makefile can be executed one at a time in the command line. Do this to verify that there are no syntactical errors in the makefile, and that each command will execute separately. Once this has been done, the next command may be checked since the object files (.o) have been generated.

Q: Why won’t my make file run?

A: Common errors:

* Spaces instead of tabs on the command lines.
* Misspelled dependencies (the text after the colons).
* On the colon lines, make sure the text there (the dependency) matches what is generated by other rules.
* Make sure rules are not commented out.
* Every .cpp file needs it own rule declaration. DO NOT MASH!
* When defining an a.out rule (the one at the top, also known as the default rule), the only dependencies listed should be the .o files generated by the other rules.
* The tab character needs to be a literal tab! The width of the tab is irrelevant.
* The a.out rule should be the first rule listed in the file.
* The rule which makes the .o files uses the g++ -c option followed by the name of the .cpp file being compiled.
* The rule which makes the executable file should only use g++ followed by the name of its .o dependencies. Nothing else goes here.
* The clean rule has no dependencies, but DOES still use the colon.