Header files and using makefile

Anna Simon

October 9, 2015

This is the way we have been writing our programs so far:

```
1  #include <iostream>
2
3  int add(int x, int y)
4   {
5       return x+y;
6   }
7
8  int main()
9  {
10      using namespace std;
11      cout << "3 + 4 = " << add(3, 4)
              << endl;
12      return 0;
13 }
```

- everything is in one file
- with a lot of functions defined the file is getting difficult to read
- if any consts are declared as global variables, they might get lost among all the function declarations

It would be much easier to read if the functions were separated into other files...

A add.cpp file:

```
1 int add(int x, int y)
2  {
3      return x+y;
4  }
```

Then in the main.cpp file:

```
1 #include <iostream>
2
3 int add(int x, int y); //forward
      declaration
4
5 int main()
6 {
7      using namespace std;
8      cout << "3 + 4 = " << add(3, 4)
           << endl;
9      return 0;
10 }
```

- now the functions are in separate file(s)
- but the list of forward declarations in the main file might get long
- plus if the functions are used in multiple files, each of them has to include the forward declarations
- which means that if the function changes (e.g., number of arguments changes) the forward declarations have to be updated in all the files...

NOTE: each .cpp file needs to have all the required header files!

So... let's put all the forward declarations into a header file!

A header file add.h:

```
1 #ifndef ADD_H
2 #define ADD_H
3     int add(int x, int y);
4 #endif
```

← preprocessor declarations (we'll talk about them in couple slides)

The add.cpp file includes add.h:

```
1 #include "add.h"
2 int add(int x, int y)
3  {
4     return x+y;
5  }
```

← "" includes a file from the current directory

And so does main.cpp:

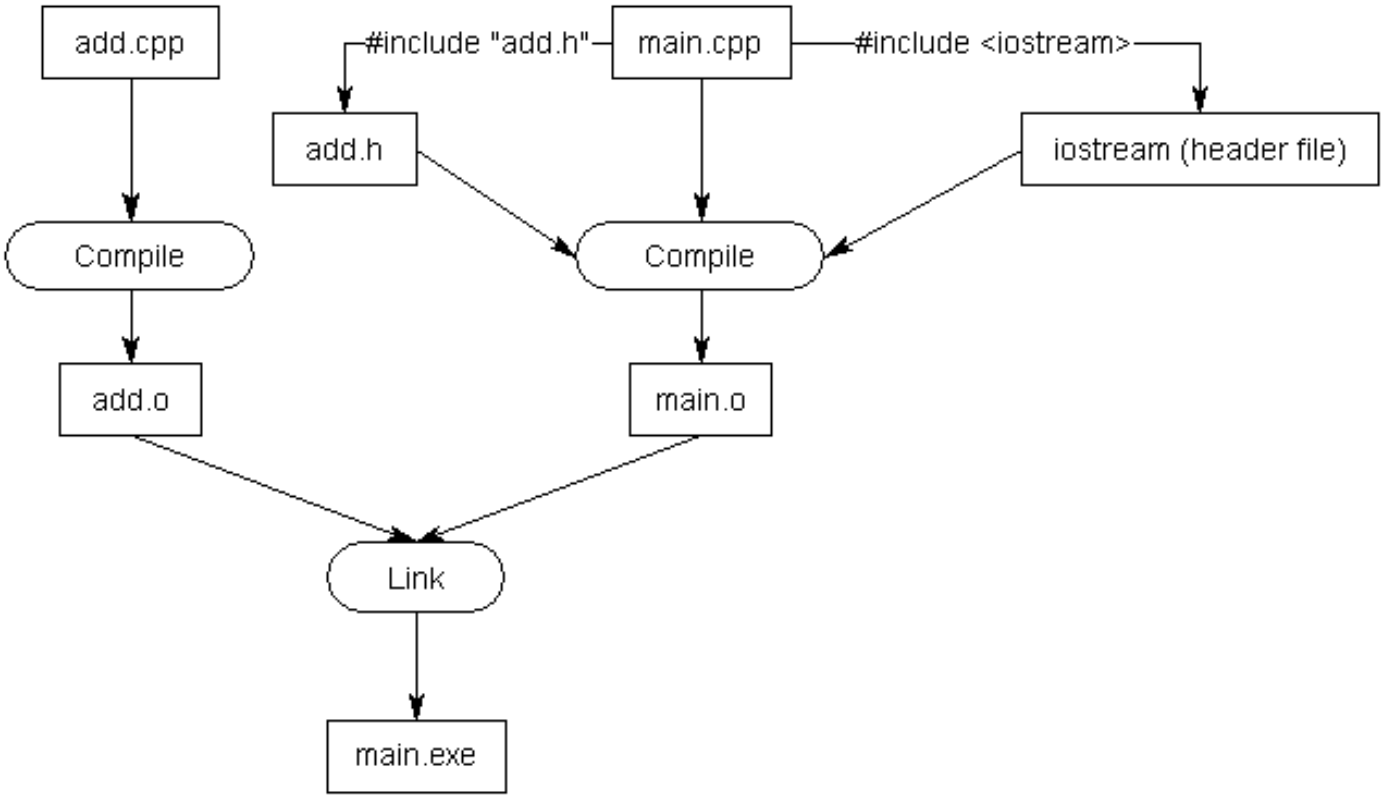
```
1 #include <iostream>
2 #include "add.h"
3
4 int main()
5 {
6     using namespace std;
7     cout << "3 + 4 = " << add(3, 4)
          << endl;
8     return 0;
9 }
```

← add.h included in main.cpp as well

But how does the compiler know where to look for the implementation of add()?

Compile: `g++ add.cpp main.cpp -o main`

alternatively: `g++ -c add.cpp main.cpp`
`g++ add.o main.o -o main`



Preprocessor declarations:

C++ preprocessor provides the ability to e.g.:

- to include files (#include)
- use conditional statements: #if, #ifdef, #ifndef, #else, #elif and #endif

In the add.h header file:

```
1 #ifndef ADD_H
2 #define ADD_H
3     int add(int x, int y);
4 #endif
```

the preprocessor declarations are used as header guards: they declare a variable ADD_H when the file is included for the first time in the code. If the header file is included again, the header guards will not allow for redeclaration of add().

- DO NOT use preprocessor declarations to declare consts!!!

When to use multiple files?

- separate function definitions (.cpp + .h)
- separate class definitions (.cpp + .h)
- separate global variables (e.g. constants) (.h)
- separate the variables that are frequently modified by the user (.h):
  - strings with input/output file locations and names
  - variables that are user specific (e.g. target mass, projectile mass, energy, charge, etc.)
  - analysis conditions (e.g. gate limits, calibration coefficients)
  - conditions for data analysis (e.g. singles or coincidence mode, switch ON/OFF some of the detectors)
  - output parameters (e.g. flags to enable/disable output types (ROOT histograms, trees, graphs) that might be useful to speed up the analysis)
  - allows for including instructions for the user which if used in an input file would make it a bit more difficult to read in

How to make the compiling process easier?
Use `make`

Simplest makefile:

```
1 #indicates that the target 'main' depends on main.cpp and add.cpp
2 #and provides instruction how to build main
3 main: main.cpp add.cpp
4       g++ -o main main.cpp add.cpp
5
6 #this will remove the executable file
7 clean:
8       rm -f main
```

NOTE: there has to be a [tab] before the g++ command, or else make will complain. In XCode use Option+Tab to create that spacing, otherwise make will see it as four spaces
Comments in Makefile begin with #

To run: `make` will use Makefile by default

`make -f specialMakefile` to use a specified file

Makefile

Makefile allows for use of variables:

```
1  #this is the compiler that will be used:
2  CXX:=g++
3
4  #this flag will print out all the warnings generated by the compiler
5  CXXFLAGS:=-Wall
6
7  #this are the source files used:
8  FILES:=main.cpp add.cpp
9
10 #this is the executable name
11 EXECUTABLES:=main
12
13 all:
14     $(CXX) $(CXXFLAGS) $(FILES) -o $(EXECUTABLES)
15
16 #this will remove the executable file
17 clean:
18     rm -rf $(EXECUTABLES)
```

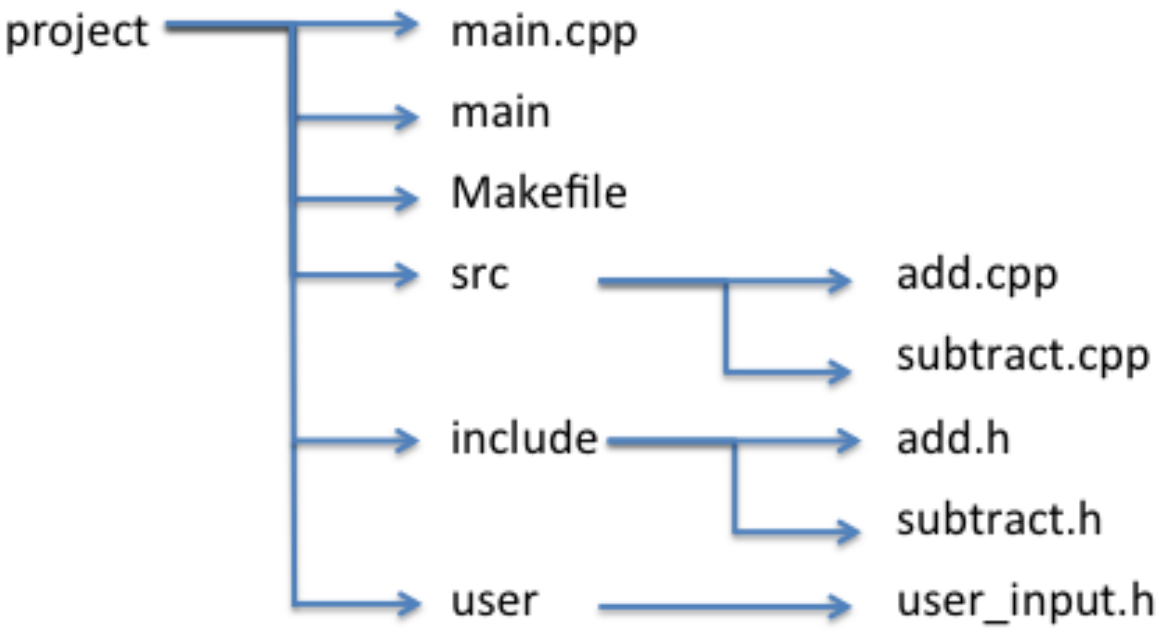
Makefile: rules for compiling and creating object files

```
1  #this is the compiler that will be used:
2  CXX:=g++
3  #this flag will print out all the warnings generated by the compiler
4  CXXFLAGS:=-Wall
5  #this are the source files used:
6  FILES:=main.cpp add.cpp
7  #this is the executable name
8  EXECUTABLES:=main
9
10 #object files (use the FILES but replace .cpp with .o)
11 OBJECTS := $(FILES: .cpp=.o)
12
13 all:
14     $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(EXECUTABLES)
15
16 add.o: add.cpp
17     $(CXX)  -c add.cpp -o add.o
18
19 main.o: main.cpp
20     $(CXX) -c main.cpp -o main.o
21
22 #this will remove the executable file
23 clean:
24     rm -rf $(EXECUTABLES)
```

Makefile: rules for compiling and creating object files: universal Makefile

```
1  #this is the compiler that will be used:
2  CXX:=g++
3  #this flag will print out all the warnings generated by the compiler
4  CXXFLAGS:=-Wall
5  #this are the source files used:
6  FILES:=main.cpp add.cpp
7  #this is the executable name
8  EXECUTABLES:=main
9
10 #object files (use the FILES but replace .cpp with .o)
11 OBJECTS := $(FILES: .cpp=.o)
12
13 all:
14     $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(EXECUTABLES)
15
16 #compile all source files and generate object files
17 # %.cpp = any file that matches the pattern
18 # $< = name of the first dependency file
19 # $@ = name of the target file
20 %.o: %.cpp
21     $(CXX) $(CXXFLAGS) -c $< -o $@
22
23 #this will remove the executable file
24 clean:
25     rm -rf $(EXECUTABLES)
```

A well organized project



```
CXX := g++
CXXFLAGS := -Wall -g -O2
#folders including all the header files
INCLUDES := -Iinclude -Iuser
#folder with all the source files
SOURCES := src
#source files
FILES := add.cpp subtract.cpp main.cpp
#this is the executable name
EXECUTABLES := main
#this are object files (take the names of sources and change the
    extensions)
OBJECTS := $(FILES:.cpp=.o)

#don't expect output files from 'clean' and 'rmobjects'
.PHONY: clean rmobjects

$(EXECUTABLES): $(OBJECTS)
    $(CXX) $(INCLUDES) $(CXXFLAGS) $(OBJECTS) -o $(EXECUTABLES)

#compile all source files and generate object files
# %.cpp = any file that matches the pattern
# $< = name of the first dependency file
# $@ = name of the target file
%.o: $(SOURCES)/%.cpp
    $(CXX) $(INCLUDES) $(CXXFLAGS) -c $< -o $@

#this will remove the executable file and objects
clean:
    rm -rf $(EXECUTABLES) $(OBJECTS)

rmobjects:
    rm -rf $(OBJECTS)
```