CSE 232, Lab Exercise 6

Partner

Choose a partner in the lab to work with on this exercise. Two people should work at one computer. Occasionally switch who is typing.

The Problem

We are going to work on two things:

1. Our first lab working with vectors
2. Understanding how we can break our program out into different files and create one executable from those files

We start with the second part. However, look for your programming tasks (there are two) to accomplish at the end of this file. Don't forget!!!

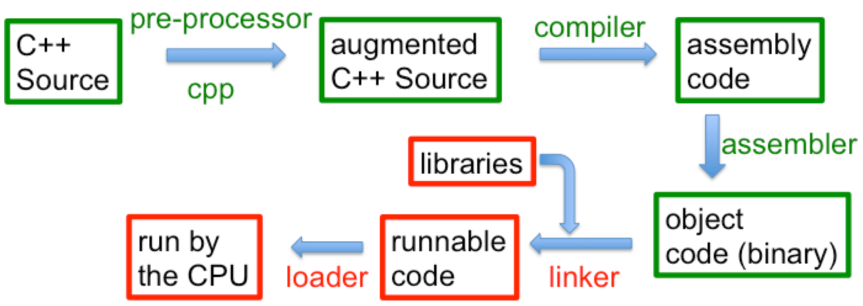
Separate Compilation of Files

Why?

Imagine that you write a really useful function (which hopefully you will today). You would like to package the function up individually so you could use it in other programs. The hard way to accomplish that would be to have to copy that function into every new program where you might want to use it. A better idea would be to place that function in its own separate file, and then compile any new program and your trusty function into one executable. That would be lovely!

How?

Remember this picture?



g++ (the underlying compiler) does all of these steps rather invisibly. To take advantage of it we need multiple files

Header Files

If we are going to define a function in one file and use the function in a main file, we are going to have to find a way to inform the main file about the **<u>types</u>** of the function. That is, we have to tell the main file:

- the function's **name**

- the **type** the function returns
- the **type** of the parameters the function uses
    - the **name** of each parameter is **unimportant**. It can be given, changed in the main function or just left out. All that matters is each parameter's **type**

If we tell the main function this information, that is enough for the C++ compiler to check that the function is being used correctly in main, and by correctly I mean that main is using all the types correctly in calling the function, even though it does not yet have available the actual function code.

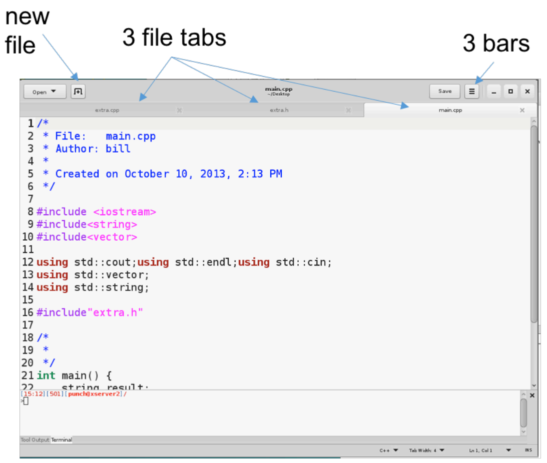
Providing this information is the job of a *header file*. Header files that users write typically end in `.h`, and are used to indicate the type information of a some C++ elements (of functions, or classes, or some other C++ thing). This header file is used by the compiler to make sure that, whoever is using this function, they are at least using the types correctly. Thus without the function itself, we can know that we followed the compiler rules and used the correct types.

Example

Make a new directory/folder, call it lab6, as you have been doing all along (either with your File Browser and the Create Folder dropdown or with the command line command mkdir).

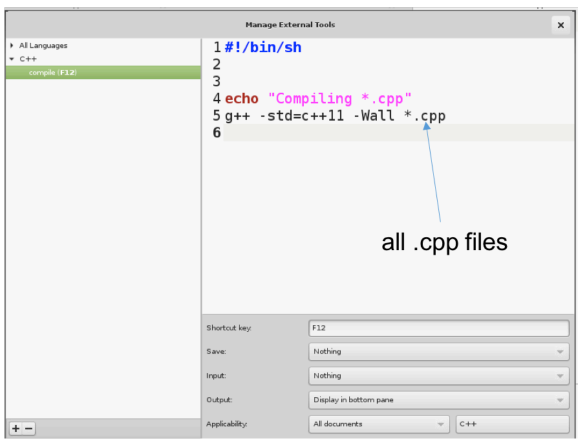
In 232 browser for lab6 are 3 files: `main.cpp, extra.cpp, extra.h` Copy these files to your desktop (easiest, right click each file, then Save As into your lab 6 directory/folder).

In gedit you can now Open -> Other Documents each of the three files. Should look something like the below, with the three tabs (for the three files) each opened.



```
/*
 * File:   main.cpp
 * Author: bill
 *
 * Created on October 10, 2013, 2:13 PM
 */

#include <iostream>
#include<string>
#include<vector>

using std::cout;using std::endl;using std::cin;
using std::vector;
using std::string;

#include"extra.h"

/*
 *
 */
int main() {
    string result;
```

```
[15:12][501][punch@xserver2]/
>
```



```
#!/bin/sh


echo "Compiling *.cpp"
g++ -std=c++11 -Wall *.cpp
```

Shortcut key: F12
Save: Nothing
Input: Nothing
Output: Display in bottom pane
Applicability: All documents, C++

To build a final executable, the build command has been modified in the lab and on the x2go server so that, if you build the project now, all the files will be compiled. You can see this by going to the "three bars" (top right), pick "Manage External Tools" and then "compile (F12)"

```
        g++ -std=c++11 -Wall  *.cpp
```

That means, it will compile all (* means all names, *.cpp means all files ending in .cpp) the `.cpp` files and build an executable.

Two warnings!

1) It's nice that by hitting F12 we compile all the files, but if you have too many files (from different projects, things you are working on temporarily etc.) it won't work. You can do it from the command if you'd prefer with a list of files. You can even name your executable using the –o modifier to be something than the dreaded `a.out`

```
g++ -std=c++11 –Wall file1.cpp file2.cpp file3.cpp –o namedExecutable.exe
```

2) We never compile a .h file. That doesn't make any sense. All a .h file provides is a list of declarations to be used by other files. It is never compiled and would not show up in the list of files to compile (shown above)

Show your TA that you got the three files, made a project, compiled and ran it.

The Files

- `extra.cpp`. It defines the function `extra` which will be used in the `main` program.
- `extra.h`. This is the header file. Notice that is only really provides the declaration of the function.
    - In the declaration, the names of the parameters are not required, only their types.
    - Note that the function declaration ***ends in a ;*** (semicolon). Don't forget!!!
    - There are some weird # statements, we'll get to that.
- `main.cpp`. This is the `main` program. Notice that it has the following statement in it:

  ```
  #include "extra.h"
  ```

  This means that the `main` program is "including" the declaration of the function so that the compiler may check the type use of `extra` by `main`. Notice the quotes. When the `include` uses quotes, it is assumed that the `.h` file is in the same directory as the other files. `include` with <> means get includes from the "standard include place". Since it is our include file, we need to use quotes for it.

Weird # in headers

Anything beginning with # is part of the pre-processor. This controls aspects of how the compilation goes. In this case, we are trying to prevent a multiple definition error. What if we wanted to use `extra` in a more than one file? Your expectation is that every file that wants to use `extra` should include the `extra.h` file for compilation to work, and you would be correct. Sort of. Remember that you cannot declare a variable more than once, and the same goes for a function. You should only declare it once. In making one executable from many files, it is possible that, by including `extra.h` in multiple files, we would declare the `extra` function multiple times for this single executable. C++ would complain. However, it would be weird to have to do this in some kind of order where only one file included `extra.h` and assume the rest would have to assume it is available.

The way around this involves the pre-processor. Basically there are three statements we care about:

```
#ifndef some_variable_we_make_up
#define some_variable_we_make_up
```

… all the ***<u>declarations</u>*** in this .h file

```
#endif
```

This means. "If the pre-processor variable we indicate (`some_variable_we_make_up)` is not defined (`#ifndef`), go ahead and define it (`#define`) for this compilation and do everything else up to the `#endif,` meaning include the declarations in the compilation. If the variable is already defined, skip everything up to the `#endif`, meaning skip the declarations"

Thus whichever file pulls in the header file first, defines the pre-processor variable and declares the function for the entire compilation. If some other file also includes the header file later in the compilation, the pre-processor variable is already defined so the declarations are not included.

Programming Task 1

Make a new project in your lab 6 directory called `splitter`. We want to add three new files to the project: `main.cpp functions.cpp functions.h`

- Make a new file (weird icon next to open) then save as `main.cpp`
- Now make the `functions.cpp` and the `functions.h` file.
- You should have three file tabs at the top now.

Function split

The split function should take in a `string` and return a `vector<string>` of the individual elements in the string that are separated by the separator character (default `' '`, space). Thus

`"hello mom and dad"` → `{"hello", "mom", "and", "dad}`

- Open `functions.h` and store the function ***<u>declaration</u>*** of `split` there. The declaration should be:

```
vector<string> split (const string &s,
                      char separator=' ');
```

- As discussed in class, default parameter values ***go in the header file only***. The default does not occur in the definition if it occurred in the declaration.
- This header file should wrap all declarations using the `#ifndef, #define, #endif` as discussed above. Make up your own variable.

- Open `functions.cpp` and write the definition of the function `split`. Make sure it follows the declaration in `functions.h`. The parameter names do not matter but the types do. Make sure the function signature (as discussed in class) match for the declaration and definition.
- You can ***compile*** `functions.cpp` (not build, at least not yet) to see if `functions.cpp` is well-formed for C++. It will not build an executable, but instead a `.o` file. The `.o` file is the result of compilation but before building an executable, an in-between stage.

Function print_vector
This function prints all the elements of `vector<string> v` to the `ostream out` provided as a **reference parameter** (it must be a reference). Note `out` and `v` are passed by reference.

- `print_vector` : Store the function `print-vector` in `functions.cpp`, put its declaration in `functions.h` It should look a lot like the below:

```
void print_vector (ostream &out, const vector<string> &v);
```

- compile the function (not build, compile) to make sure it follows the rules.

Function main
The `main` function goes in `main.cpp`.

- In main.cpp make sure you `#include "functions.h"` (note the quotes), making those functions available to main. Write a `main` function that:
- The operation of main is as follows:
    - prompts for a string to be split
    - prompts for the single character to split the string with
    - splits the string using the `split` function which returns a vector
    - prints the vector using the `print_vector` function
- compile (not build) main to see that it follows the rules

Build the executable
Select the `main.cpp` tab of geany and now build the project. An executable called main should now be created which you can run.

Assignment Notes

1. Couple ways to do the split function
    1. `getline`
        i. `getline` takes a delimiter character as a third argument. In combination with an input string stream you can use `getline` to split up the string and `push_back` each element onto the vector.
        ii. example `getline(stream, line, delim)` gets the string from the stream (`istream`, `ifstream`, `istringstream`, etc.) up to the end of the line or the delim character.
    b. string methods `find` and `substr`. You can use the `find` method to find the delim character, extract the split element using `substr`, and `push_back` the element onto the vector.
    2. Default parameter value. The default parameter value needs to be set at **declaration time**, which means that the default value for a function parameter should be in the header file (the declaration). If it is in the declaration, it is not required to be in the definition, and by convention should not be.

Programming Task 2

Let's play a little more with vectors. Let's write a function that can add and subtract the elements of two vectors.

1. Create a new directory in your `lab6` directory calld `vec_ops`
    a. three files again: `main.cpp, functions.cpp, functions.h`
1. The `functions.h` has the following single function signature:

```
a. vector<long> vector_ops(const vector<long>& v1,
                          const vector<long>& v2,
                          char op)
```

2. In `functions.cpp` write `vector_ops` to do the following:
    a. find the shorter of the two vectors
    b. for each element in both vectors, up to the length of the shorter vector
        i. if the op is `'+'`, add each element into a new vector (to be returned)
        ii. if the op is `'-'`, subtract the shorter vector element from the longer vector element into a new vector.
        iii. after the operation, copy the remaining elements from the longer vector in the result vector
        iv. if op is something else, return an empty `vector<long>`
3. Write a `main.cpp` and test your function by sending in two vectors and printing the result