Multiple files

Comp Sci 1570 Introduction to C++

Missouri S&T | Computer Science

Outline

Outline

Monolithic vs Modular

One file before

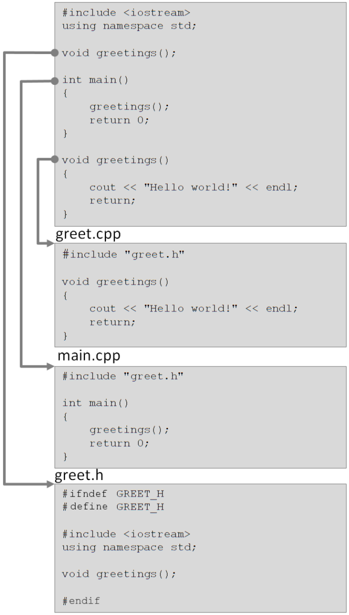
- system includes
- prototypes
- main driver function
- function definitions

Multiple files now

- main driver file
- prototypes header file(s)
- functions definition implementation source file(s)

Outline

Multiple files: diagram and source files: greet*.*



```
#include <iostream>
using namespace std;

void greetings();

int main()
{
    greetings();
    return 0;
}

void greetings()
{
    cout << "Hello world!" << endl;
    return;
}
```

greet.cpp

```
#include "greet.h"

void greetings()
{
    cout << "Hello world!" << endl;
    return;
}
```

main.cpp

```
#include "greet.h"

int main()
{
    greetings();
    return 0;
}
```

greet.h

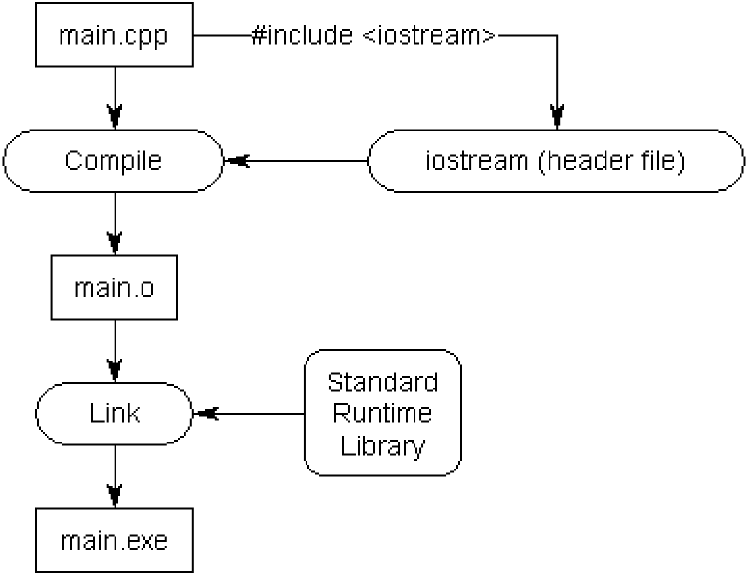
```
#ifndef GREET_H
#define GREET_H

#include <iostream>
using namespace std;

void greetings();

#endif
```

What does including iostream do at compile time?

```
#include <iostream>
int main()
{
  std::cout << "Hello,␣world!" << std::endl;
  return 0;
}
```

including iostream

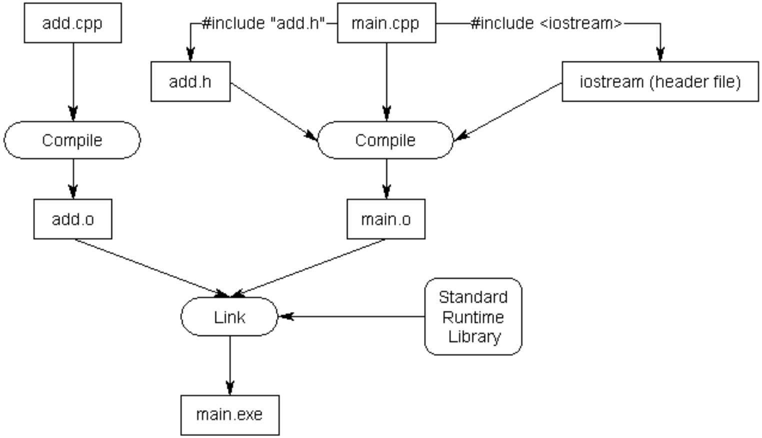
Header enables calling, iostream is linked, not copied into main



Example: add

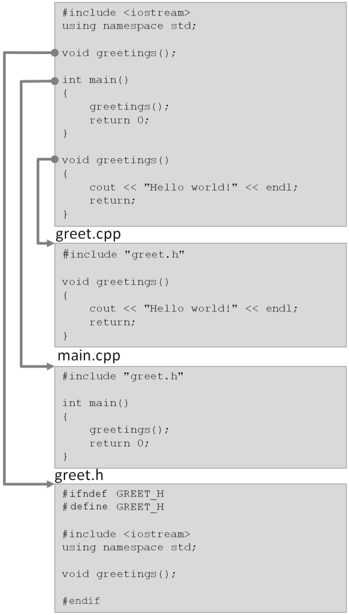
- See files: add.cpp, add.h, add_main.cpp

Multiple files



Header allows separate compiling, and linking after, for speedy compile and re-compile. Whereas including the add.cpp directly in main would work, it would copy the entire add.cpp into main at compile time, increasing compile time, especially for big projects

Multiple files



```
#include <iostream>
using namespace std;

void greetings();

int main()
{
    greetings();
    return 0;
}

void greetings()
{
    cout << "Hello world!" << endl;
    return;
}
```

greet.cpp

```
#include "greet.h"

void greetings()
{
    cout << "Hello world!" << endl;
    return;
}
```

main.cpp

```
#include "greet.h"

int main()
{
    greetings();
    return 0;
}
```

greet.h

```
#ifndef GREET_H
#define GREET_H

#include <iostream>
using namespace std;

void greetings();

#endif
```

Outline

Example main file

```
// Author: Clayton Price
// File: main.cpp

#include <iostream>
#include "treeFarm.h"

using namespace std;

int main()
{
  cout << tree_price();

  return 0;
}
```

Example implementation file

```
// Programmer: Clayton Price date: 10-1-10
// File: treeFarmFuncs.cpp
// Purpose: This file contains the function
// definitions for the tree farm program.

#include "treeFarm.h"
using namespace std;

float tree_price(const int numtrees, const string treetype)
{
  // body of implementation here
}
```

Example header file

```
// treeFarm.h in uppercase follows convention
#ifndef TREEFARM_H

// Can be anything unique,
// but keeping this convention is nice
#define TREEFARM_H

// Programmer: Clayton Price date: 10-1-10
// File: treeFarm.h
// Purpose: this file contains the prototypes
// Constants for the tree farm program

// This constant states the wt per board foot
const float OAK_DENSITY = 4.25; // pounds per
// more header here...

#endif
```

Best practices for creating your own header files

- Always include header guards (a kind of preprocessor directives)
- Do not define variables in header files unless they are constants. Header files should generally only be used for declarations.
- Do not define functions in header files.
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h.
- Give your header files the same name as the source files they're associated with (e.g. grades.h goes with grades.cpp).
- Try to minimize the number of other header files you #include in your header files. Only #include what is necessary.
- Do not #include .cpp files; it works, but why not?

Outline

Preprocessor Directives for Header Files

Put at the top

```
#ifndef MYFILE_H
#define MYFILE_H

//header content goes here

#endif
```

at the bottom

Outline

Preprocessor

- The preprocessor is perhaps best thought of as a separate program that runs just before the compiler when you compile your program.
- When the preprocessor runs, it simply scans through each code file from top to bottom, looking for directives.
- Directives are specific instructions that start with a # symbol and end with a newline (NOT a semicolon).
- There are several different types of directives, which we will cover below.
- The preprocessor is not smart; it does not understand C++ syntax; rather, it simply manipulates text before the compiler runs.
- The output of the preprocessor is then sent to the compiler.
- Note that the preprocessor does not modify the original code files in any way; rather, all text changes made by the preprocessor happen temporarily in-memory.

Includes

- When you #include a file, the preprocessor copies the contents of the included file into the including file at the point of the #include directive. This is useful when you have information that needs to be included in multiple places (as forward declarations often are).
- The #include command has two forms:
  - #include < *filename* > tells the preprocessor to look for the file in a special place defined by the operating system where header files for the C++ runtime library are held. You'll generally use this form when you're including headers that come with the compiler (e.g. that are part of the C++ standard library).
  - # include "filename" tells the preprocessor to look for the file in the directory containing the source file doing the #include. If it doesn't find the header file there, it will check any other include paths that you've specified as part of your compiler/IDE settings. That failing, it will act identically to the angled brackets case. You'll generally use this form for including your own header files.

Duplicate variable definition

```
#include <iostream>
int main()
{
  // this is a definition for identifier x
  int x;

  // compile error: duplicate definition
  int x;

  return 0;
}
```

Duplicate function definition

```
#include <iostream>

int foo()
{
  return 5;
}

// compile error: duplicate definition
int foo()
{
  return 5;
}

int main()
{
  std::cout << foo();
}
```

Double include mistake

math.h:

```
int getSquareSides(){
    return 4;
}
```

geometry.h:

```
#include "math.h"
```

main.cpp:

```
#include "math.h"
#include "geometry.h"
int main(){
    return 0;
}
```

Double include mistake

After resolving all of the #include, main.cpp:

```
int getSquareSides()  // from math.h
{
    return 4;
}

int getSquareSides() // from geometry.h
{
    return 4;
}

int main()
{
    return 0;
}
```

Preprocessor directives

- Header guard preprocessor directives will prevent the compiler from trying to create multiple definitions for the functions and constants and anything else in the header if this header is included in a translation unit multiple times.
- Even the system header files have them.
- Do this for every header file you create.
- The identifier convention is to use all uppercase versions of the file names (as demonstrated in the example above).
- Incidentally, ifndef stands for “if not defined”.
- So, you can read those directives as “if not defined, define”.

Header guard conditional compilation directives

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations and definitions here

#endif
```

See example: square*.* source files

Outline

User Includes and Compiling with multiple files

When you compile a program split into multiple files, normally, the command would be

```
g++ prog.cpp -o my_prog
```

The source code is prog.cpp and the executable's name will be my_prog. We wish to compile the previous tree farm program:

```
g++ treeFarm.cpp treeFarmFuncs.cpp -o trees
```

Provide the names of all compilable files as the source code (italicized in the example above). Alternatively, you may use the wildcard character, *

```
g++ *.cpp -o trees
```

This command will pick up all files in the current directory with a .cpp extension. So, if you use this method, it is vitally important to put your programming projects in their own directory. You don't want to have more than one main function in your directory.

Code::Blocks with multiple files

To use multiple in this way with Code:Blocks, you must create a project from and empty project file, and add the files to it. Remember however, that you should compile and run on the campus Linux computers before submission, since environments can differ.