C++

Preprocessor and linking

Michael Burrell

January 9, 2020

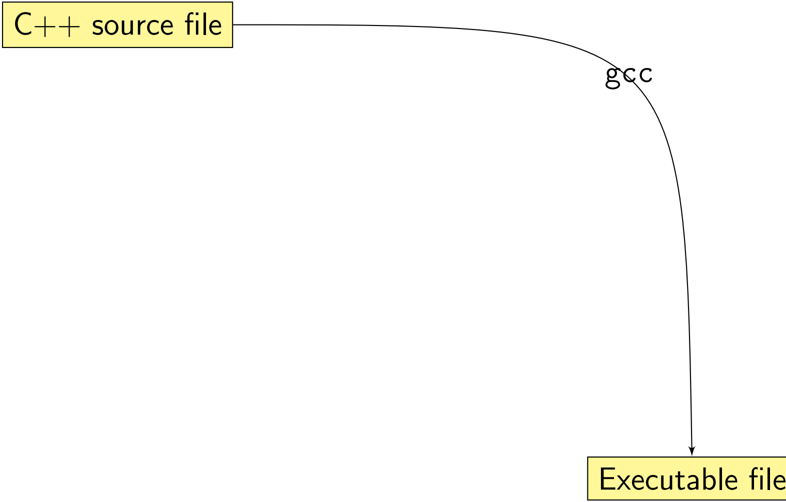
Textbook readings

- Chapter 1
- Chapter 2

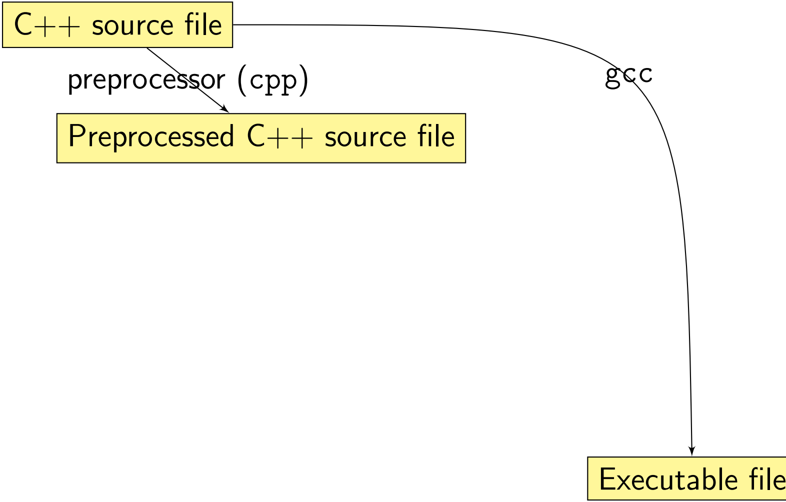
Goals for this set of slides

- Understand how `#include` actually works
- Understand why we need header files
- Understand how an executable file is formed
- Describing the step-by-step process of turning C++ source code into an executable

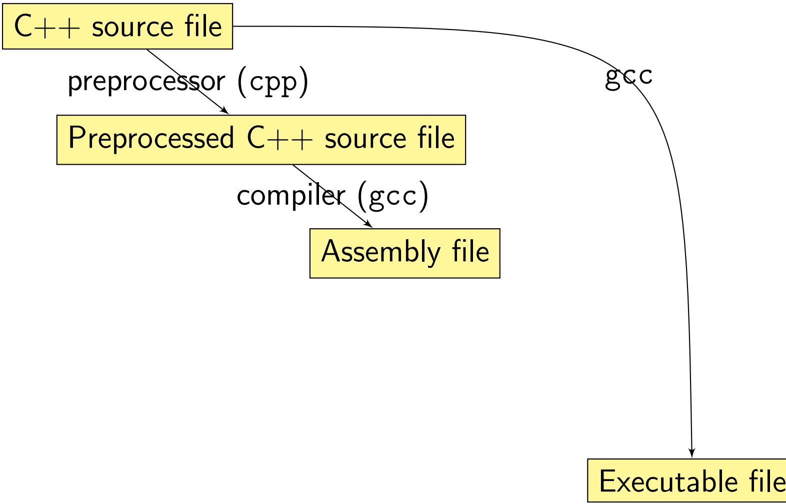
In the Unix world



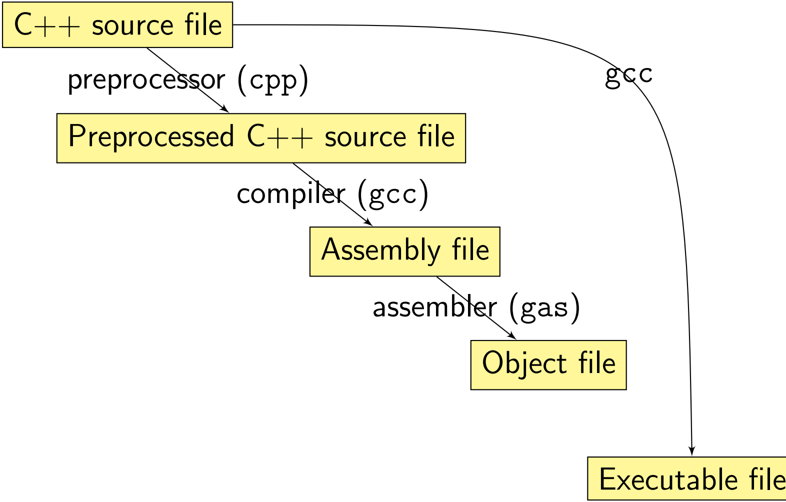
In the Unix world



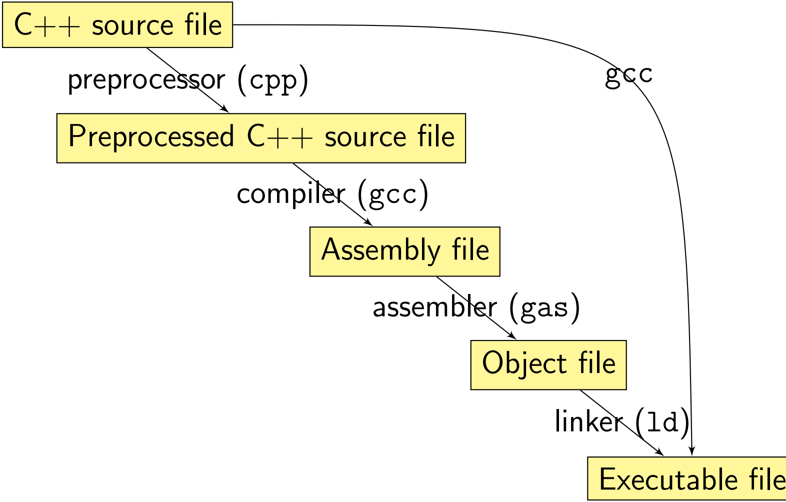
In the Unix world



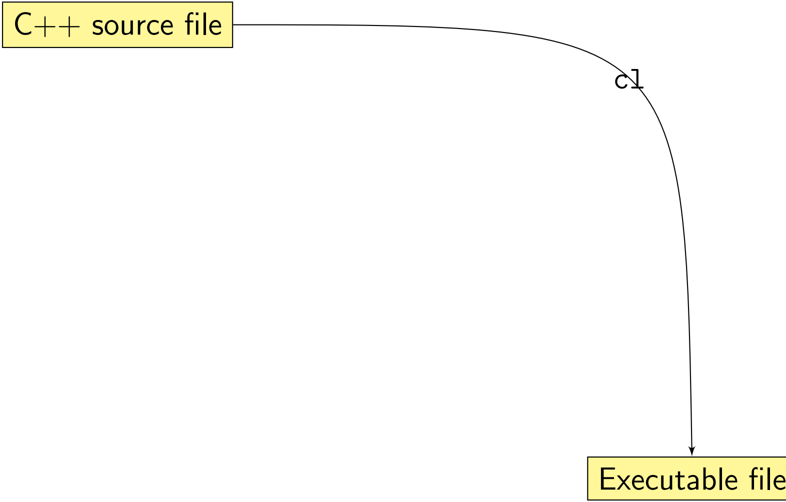
In the Unix world



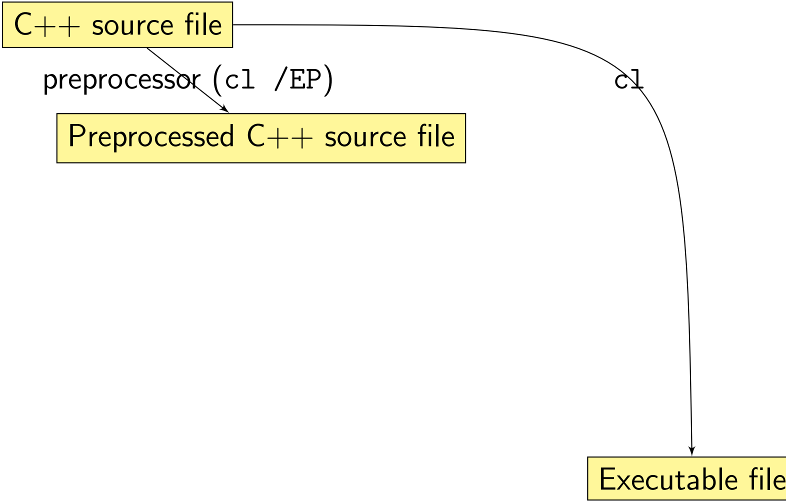
In the Unix world



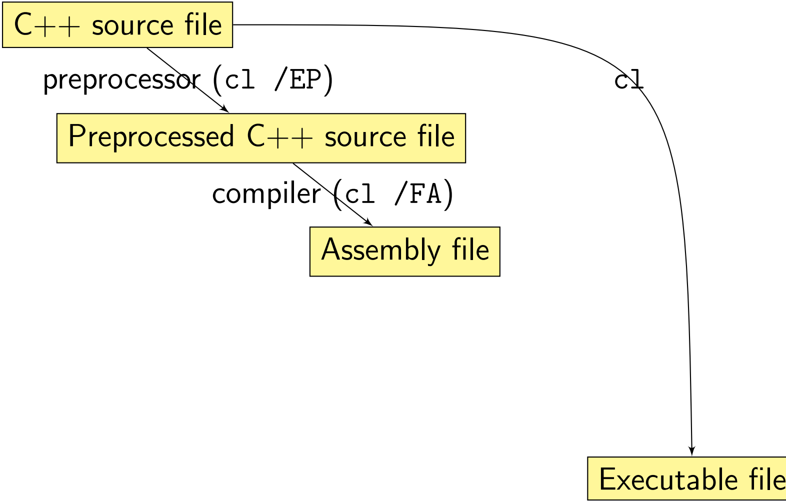
In the Visual Studio Windows world



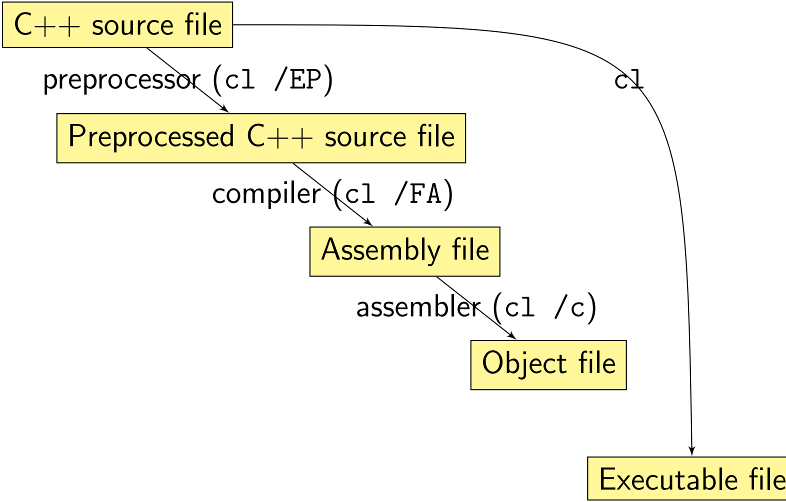
In the Visual Studio Windows world



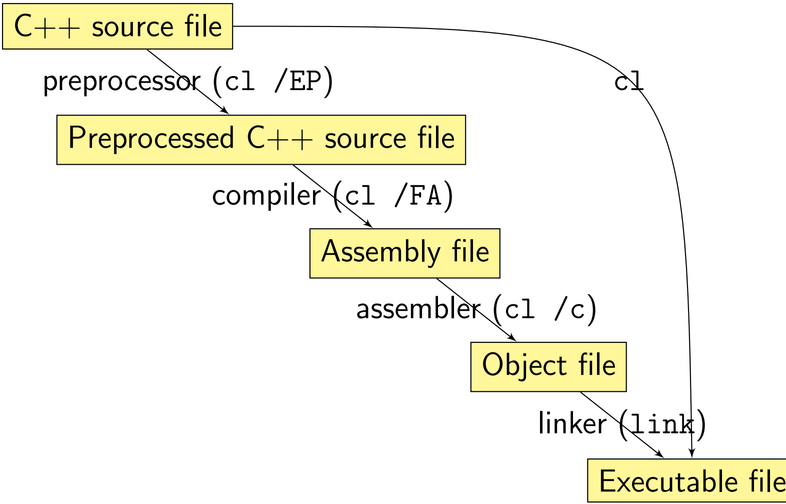
In the Visual Studio Windows world



In the Visual Studio Windows world



In the Visual Studio Windows world



Compilation procedure

- Modern compilers (like `gcc` and `cl`) do all 4 of these steps for you automatically
- You can tell the compiler to only do one of the steps, though, which is sometimes required
- To do preprocessing, run `cl /EP`
- To do compiling (but not assembling), run `cl /FA`
- To do assembling (but not linking), run `cl /c`

Preprocessing

- The C++ preprocessor begins on lines that begin with a # sign (we've seen these before: where?)
- The C++ preprocessor is **only** capable of **textual substitution**
- The C++ preprocessor reads, as input, a C++ file and generates, as output, a C++ file

Macros

`#define`

The `#define` preprocessor directive tells the preprocessor to define a new textual substitution token. E.g.:

```
#define PI 3.14
```

Macros

#define

The #define preprocessor directive tells the preprocessor to define a new textual substitution token. E.g.:

```
#define PI 3.14
```

#define

Let's try out an example that uses this #define and see what happens when we run the preprocessor.

Macros

Macros with parameters

You can get slightly more advanced macros by including parameters:

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

Macros

Macros with parameters

You can get slightly more advanced macros by including parameters:

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

More advanced yet

You can actually do a surprisingly advanced amount of stuff using even more advanced preprocessor macros (##, the "paste operator", as well as the "stringify operator" and `#error` and `#line` directives).

We're going to skip over all of these advanced usages.

Your requirements for familiarity with the preprocessor in C++ are very low in this course.

The #include directive

- `#include` is just a textual substitution. It tells the C preprocessor, literally "take another file and paste it in this file"
- `#include <whatever>` looks for the file `whatever` in "a standard location" (on Unix systems, this is `/usr/include`)
- `#include "whatever"` looks for the file `whatever` in the current directory (the same place the .cpp file is)

Placement of `#include` directives

- In C++, all symbols must be declared *before* the are used
- If an included file (like `iostream`) declares symbols we need (like `cout`), we need to include that *before* it used
- By convention, we usually place all of our `#include`s at the top of the file
- Because the preprocessor is just dumbly doing a copy-and-paste, you are allowed to put it anywhere (*anywhere*) you want

stdafx.h

1. #include is just a literal textual copy-and-paste
2. Many of the standard header files are thousands (or tens of thousands) of lines long
3. The same standard header files (like `iostream`) often get included in many .cpp files in the same project
4. Each .cpp file is compiled separately

stdafx.h

1. #include is just a literal textual copy-and-paste
2. Many of the standard header files are thousands (or tens of thousands) of lines long
3. The same standard header files (like `iostream`) often get included in many `.cpp` files in the same project
4. Each `.cpp` file is compiled separately
5. This all adds up to: *parsing standard header files can take a lot of time on large builds*

stdafx.h

- To mitigate this and speed up build times, many compilers have started offering *pre-compiled header files*
- This allows the compiler to *parse* the header file and then store an internal (compiler-specific, binary) representation of that parse tree somewhere
- Then the header file only has to be parsed once
- `stdafx.h` used to be a (IMHO, poorly designed) mechanism to make this work in Visual Studio
- Only old versions of Visual Studio use it
- Don't worry too much about it

Function prototypes

- In order for one .cpp file to call a function that exists in another .cpp file, it *should* know that that function exists
- This is not a *requirement* in C++ (the C++ compiler will only give a warning if you try to to call a function it doesn't know exists)
- With some symbols (like objects, variables, or constants), it is an error, though
- In our code, whenever we're calling a function in a different .cpp file, we're going to include a function prototype to let the the C++ compiler know it exists

Multiple files

file1.cpp

```
int foo(int x)
{
    return x * 2;
}
```

`file1.cpp` is where we're going to keep the *definition* of a function. This is the complete definition of `file1.cpp` (no includes or `main` function needed)

Multiple files

file1.cpp

```
int foo(int x) {
    return x * 2;
}
```

file2.cpp

```
int foo(int);
int main() {
    return foo(3);
}
```

`file2.cpp` is where we *call* `foo`. This is also a complete definition. It does not require any includes. It *should* have a *function prototype* for `foo`.

Prototypes

- You should always have a function prototype in scope when calling a function that is not defined in the same `.cpp` file (or is defined lower down)
- Prototypes for functions look like:
  - E.g., `int foo(int, double x);`
  - The return type, name, and types of parameters are required
  - The names of the parameters are optional

Putting prototypes in headers

- We often find it more convenient to put function prototypes in header files (`.h` files) instead of `.cpp` files
- This way, every `.cpp` file (in case we have many of them) gets exactly the same prototype for a particular function
- It becomes the primary way to communicate with other `.cpp` files what symbols are available in the project

Spreading functions across files

- If a function is in a different `.cpp` file, the C compiler does not care *which* `.cpp` file it's in
- When it generates a `.obj` file (`.o` file in Unix), it will mark that function as being "undefined" (defined in a different file)
- It's the linker's responsibility, when linking all of the `.obj` files together, to figure out which function is defined in which `.obj` file

External libraries

- Because so many different applications need to do the same things, operating systems allow the notion of a *library*
- On Windows, libraries are called .dll files; on Linux, libraries are called .so files; on OS X, libraries are called .dylib files
- A library is just a bunch of `.obj` files package up together into a "super" `.obj` file
- A 3rd-party library will also need to ship with `.h` files so that we can use the symbols when compiling our code

Symbol visibility

- By default, in C++, every function that we write is global
- This means that if we write two functions with the same name, in different files, there will be a conflict
- The linker will be unable to link all the `.obj` files together into an executable

The `static` keyword

- Defining a function with the `static` keyword means that that function cannot be referenced from outside the current .cpp file
- The linker will be able to link together multiple `.obj` files that contain functions with the same name, as long as they're static
- It's good practice to make your functions `static` unless you think they will need to be used from different .cpp files

Namespaces

- Another solution to this problem of conflicting names in C++ is define our own namespace
- We will discuss namespaces later in the course

Conclusion

- We understand how the compiling process is broken up into preprocessing, compiling, assembling, and linking
- Lines with # are not C++ lines: they're lines for a C++ preprocessor that does textual substitutions
- We can make header files to help coordinate between different .cpp source files
- All of this is requires a little more work than in Java