Preprocessor and linking

C++

Michael Burrell

March 5, 2024

Textbook readings

Chapter 1 — 1.1

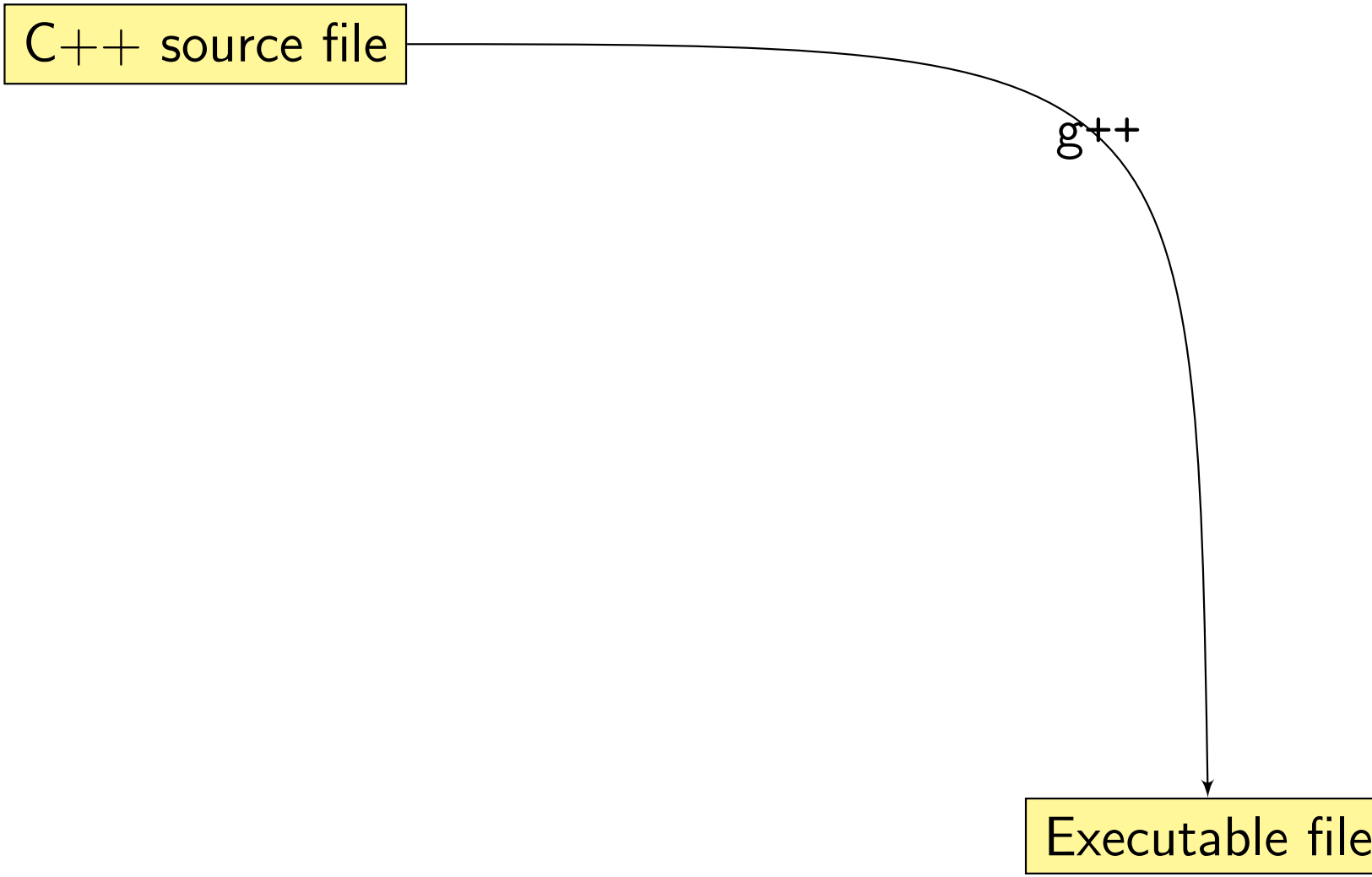
Chapter 2 — 2.6.3

Chapter 6 — 6.1

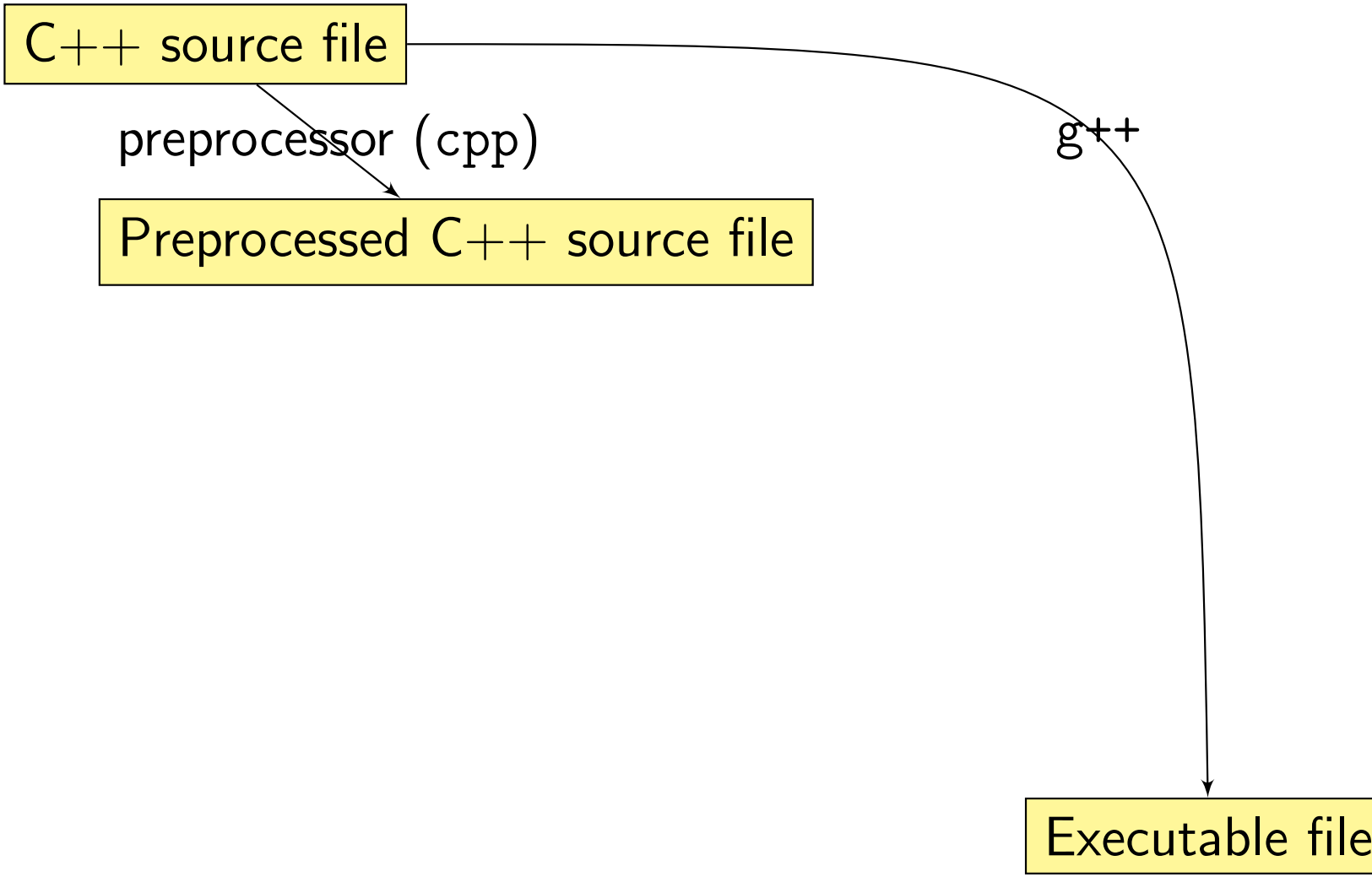
Goals for this set of slides

- 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

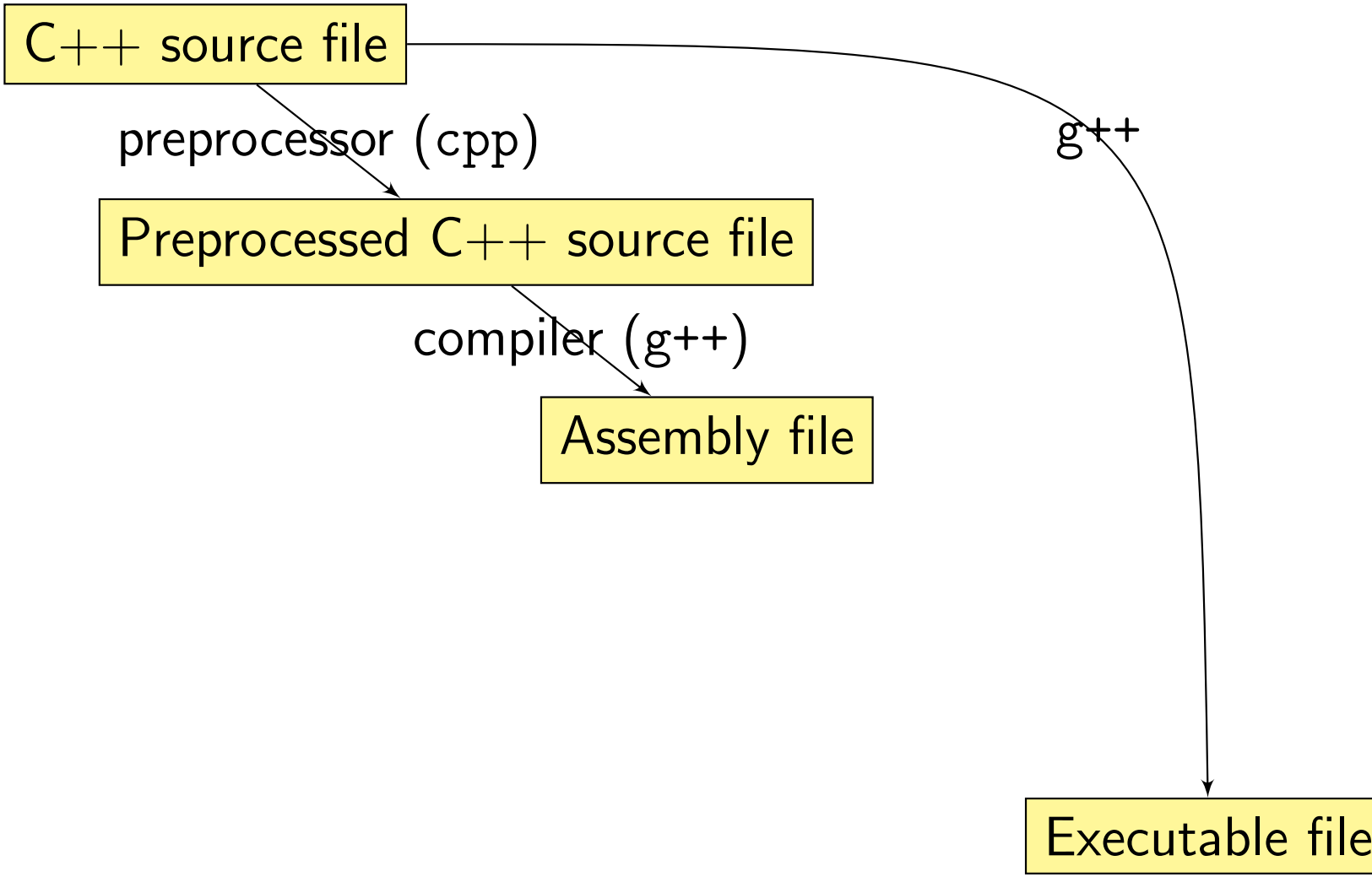
Compiling C++ code



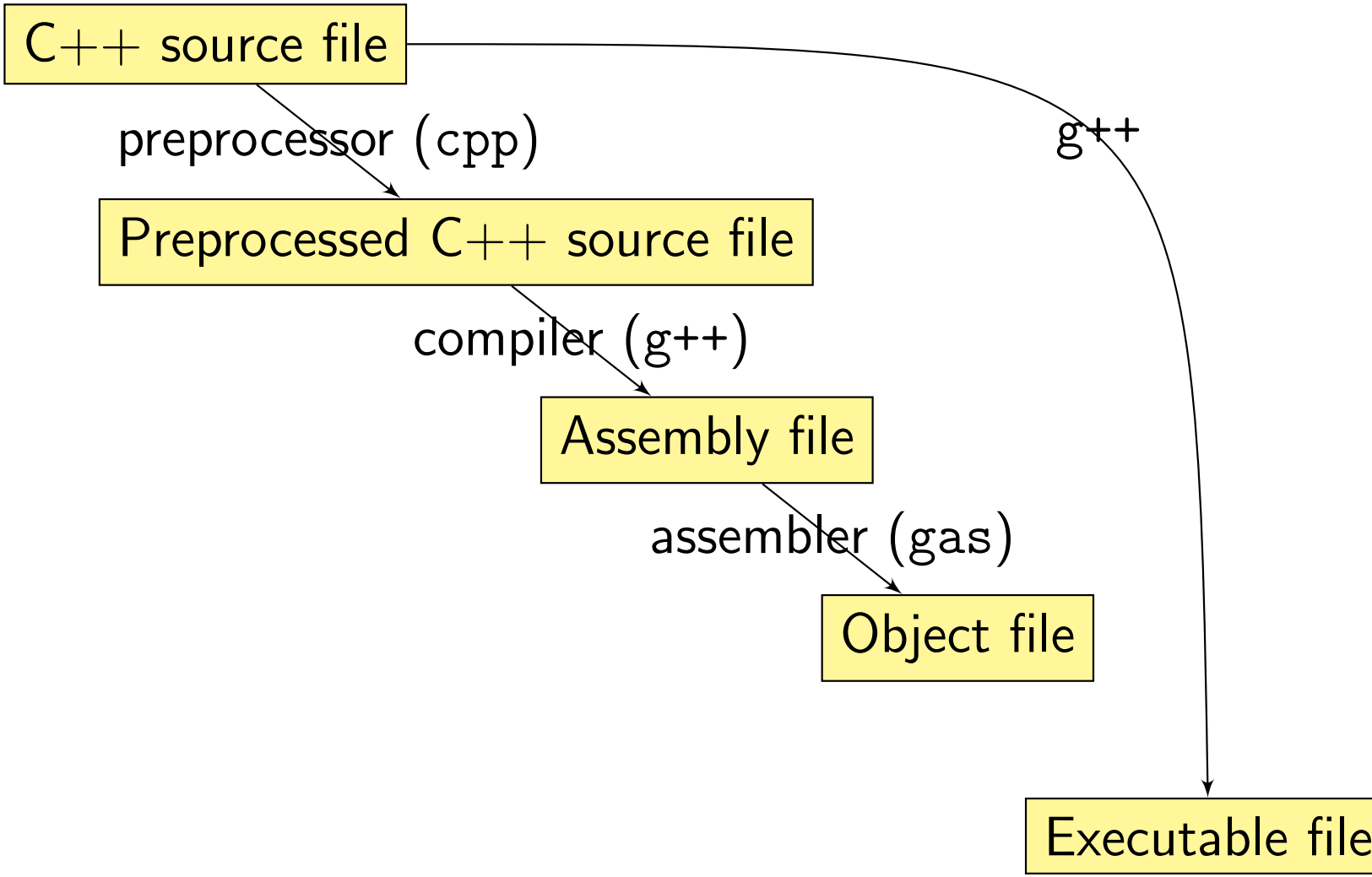
Compiling C++ code



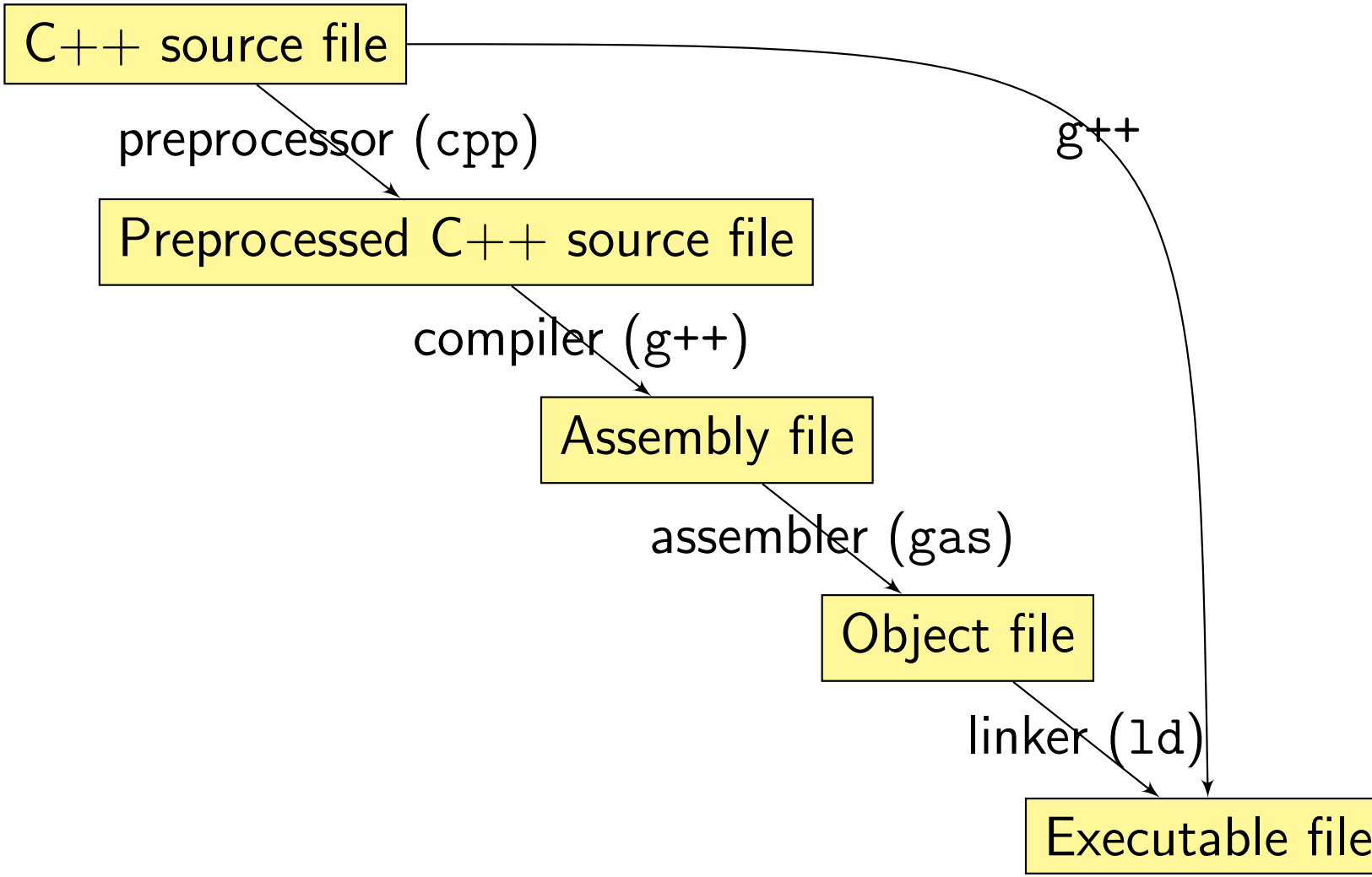
Compiling C++ code



Compiling C++ code



Compiling C++ code



Compilation procedure

- Modern compilers (like g++ and `clang`) 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 g++ `-E`
- To do compiling (but not assembling), run g++ `-S`
- To do assembling (but not linking), run g++ `-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

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.

Macros and meta-programming

- Because the C++ preprocessor doesn't understand C++ (and just copies and pastes), the arguments don't have to be expressions or values
  - They can be *any* type of text
- This makes them very valuable with repetitive types of code like automated testing

Macros and meta-programming

- Because the C++ preprocessor doesn't understand C++ (and just copies and pastes), the arguments don't have to be expressions or values
  - They can be *any* type of text
- This makes them very valuable with repetitive types of code like automated testing
- Imagine we had a function we wanted to test...repetitively...for a lot of different values

Macros with testing


```
1  bool is_palindrome(string);
2  int main() {
3  #define CHECK(expr, expected) \
4          if ((expr) != (expected)) { \
5              cerr << #expr << " failed on line " <<
                   __LINE__ << endl; \
6          }
7      CHECK(is_palindrome("radar"), true)
8      CHECK(is_palindrome("trampoline"), false)
9      // ...
10 }
```

1. This code will generate a bunch of `if` statements (one for each `CHECK`)
2. When defining a multi-line macro, `\` is needed at the end of each line (except the last line)

Macros with testing


```
1  bool is_palindrome(string);
2  int main() {
3  #define CHECK(expr, expected) \
4          if ((expr) != (expected)) { \
5              cerr << #expr << " failed on line " <<
                   __LINE__ << endl; \
6          }
7      CHECK(is_palindrome("radar"), true)
8      CHECK(is_palindrome("trampoline"), false)
9      // ...
10 }
```

3. #expr "stringifies" the expr parameter (puts double quotes around it)
4. __LINE__ is replaced by the line of code where the macro was invoked

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 .cc 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 `#includes` 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

Guards

- `.cc` files include `.h` files
- `.h` files themselves can (and often do) include other `.h` files
- This can easily lead to situations where a `.h` file ends up getting (indirectly) included multiple times
- This can cause "multiple definition" errors in the compiler or linker
- A solution to this is to put *guards* in all of your `.h` files

Guard example

math-funcs.h

```
1 #ifndef MATH_FUNCS_H
2 #define MATH_FUNCS_H
3 double square(double);
4 #endif MATH_FUNCS_H
```

geometry.h

```
1 #ifndef GEOMETRY_H
2 #define GEOMETRY_H
3 #include "math-funcs.h"
4 double hypoteneuse(double, double)
5 #endif
```

Guard example

main.cc

```
1 #include "math-funcs.h"
2 #include "geometry.h"
3 // ...
```

Guards

- Guards are a convention commonly followed in C and C++ headers as:

  `#ifndef` *xxx* is the first line (where *xxx* is any name)

  `#define` *xxx* is the second line

  `#endif` is the last line

- The first time the `.h` file is included by any `.cc` file, the `ifndef` (if-not-defined) is true, and the file is read normally
- Each subsequent time it is included, the `ifndef` is false, and so the contents of the file are skipped

Function prototypes

- In order for one `.cc` file to call a function that exists in another `.cc` file, it *should* know that that function exists
- In our code, whenever we're calling a function in a different `.cc` file, we're going to include a function prototype to let the C++ compiler know it exists

Multiple files

file1.cc

```
1 int foo(int x)
2 {
3     return x * 2;
4 }
```

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

Multiple files

file1.cc

```
1 int foo(int x) {
2     return x * 2;
3 }
```

file2.cc

```
1 int foo(int);
2 int main() {
3     return foo(3);
4 }
```

`file2.cc` 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 `.cc` 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
  - Ends in a `;` semicolon (*not* curly braces)

Putting prototypes in headers

- We often find it more convenient to put function prototypes in header files (`.h` files) instead of `.cc` files
- This way, every `.cc` 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 `.cc` files what symbols are available in the project

Spreading functions across files

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

No code in .h files

- Generally, we do *not* include code in .h files
- Doing so would require that code to duplicated in *every* .o file for which the corresponding .cc file includes it
- This restriction will be relaxed a little in the future when we talk about classes

Making executables with multiple source files

- The `make` tool implicitly knows how to *compile* C++ code
  - Beyond defining the `CXXFLAGS` variable, we don't have to give it any direction for turning `.cc` files into `.o` files
- However, `make` does not know which files to *link*
- We need to give it a list of `.o` files to link together

Makefile for one file


```
1 SANFLAGS = -fsanitize=address
2 CXXFLAGS += -Wall -g -std=c++17 -pedantic $(SANFLAGS)
3 LDFLAGS += $(SANFLAGS)
4 main:  main.o
5     $(CXX) -o $@ $^ $(LDFLAGS)
6 .phony: clean
7 clean:
8     $(RM) main *.o
```

Here's a standard Makefile based around a single `.cc` file (`main.cc`)

Makefile for two files


```
1 SANFLAGS = -fsanitize=address
2 CXXFLAGS += -Wall -g -std=c++17 -pedantic $(SANFLAGS)
3 LDFLAGS += $(SANFLAGS)
4 main:  main.o someotherfile.o
5     $(CXX) -o $@ $^ $(LDFLAGS)
6 .phony: clean
7 clean:
8     $(RM) main *.o
```

Here's a standard Makefile based around two `.cc` files (`main.cc` and `someotherfile.cc`)

Adding more .cc files

- For simple projects (like we're doing in this course), adding new `.cc` source files just means adding another `.o` file as a dependency
- Once the dependencies for your Makefile rule include all the necessary `.o` files, `make` will figure out the rest

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 `.o` 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 `.cc` file
- Functions/variables marked as `static` have *file scope*
- The linker will be able to link together multiple `.o` 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 `.cc` files

Namespaces

- Another solution to this problem of conflicting names in C++ is define our own namespace
- This is not a solution that we will explore in this course
- See `https://www.cplusplus.com/doc/oldtutorial/namespaces/` if you would like information on more advanced usages of namespaces in C++

Makefiles

- Running the commands to compile code manually is not good
  - Tedious
  - Error-prone
- `make` is an old (older than C++) build utility to compile for you

Makefiles

`make` requires the presence of a `Makefile`, which:

- Sets compiler flags and options
- Describes dependencies between different files
- Describes actions to transform input files into output files

Makefiles

- Note: `Makefiles` do *not* describe the full sequence of building
- The `make` utility infers this automatically
- `make` will skip steps that it considers unnecessary
  - E.g., it will not recompile source files which have not been changed

Variables


```
1 SANFLAGS = -fsanitize=address
2 CXXFLAGS += -Wall -std=c++14 -g $(SANFLAGS)
```

- The `Makefile` allows you to define (using =) or add onto (using +=) variables
- By convention, variables in `Makefiles` are in ALLCAPS
- Variables are commonly used to set compiler options
- Special, pre-defined variables include `CXXFLAGS` (C++ compiler options), `CFLAGS` (C compiler options), `LDFLAGS` (linker options)

Variable scope

- Variables in `Makefiles` always have global scope
- Only set things in variables that you want to apply to *all* situations
- E.g., add to `CXXFLAGS` options that you want to apply for *all* C++ compilations

Explicit rules


```
1 my-program: main.o another.o
2     $(CXX) -o $@ $^ $(LDFLAGS)
```

- An explicit rule describes how to create a particular file
- It begins with the name of a *target* (the file to create) to the left of a :
- After the : is a list of *dependencies* (input files)

Explicit rules


```
1 my-program: main.o another.o
2     $(CXX) -o $@ $^ $(LDFLAGS)
```

- After the first line of the rule come any number of commands
- Each command must be indented with a tab character (*not* space characters)
- The commands are executed in the terminal/console and should describe how to build the target file

Explicit rules


```
1 my-program: main.o another.o
2     $(CXX) -o $@ $^ $(LDFLAGS)
```

- We can make reference to variables with `$(XXXXX)` syntax
  - The pre-defined `CXX` variable is the name of the C++ compiler
- `$@` is a special variable meaning "the name of the target of this rule" (`my-program`)
- `$^` is a special variable meaning "the names of all of the dependencies of this rule"

Implicit rules

- A very observant student will note that we used `.o` files (*not* `.cc` files) as dependencies for the previous rule
- We don't need to *explicitly* say how to compile `.cc` files into `.o` files
- `make` has some built-in implicit rules for common tasks
  - Such as compiling `.cc`/`.cpp` files into `.o` files
- You can define your own implicit rules, but that is beyond the scope of this course

Example

Class exercise

Let's write a project which will calculate integer $\log_2$ of another integer, using 2 source files to do it.

Header file dependencies

- If we change a `.cc`/`.cpp` file, `make` will be smart enough to know the `.o` file needs to be recreated
- However, if we change a `.h` file, `make` *doesn't* know which files need to be recompiled
  - Sometimes important data types can change, requiring recompilation!

Manual header file dependencies


```
1 SANFLAGS = -fsanitize=address
2 CXXFLAGS += -Wall -std=c++17 -g $(SANFLAGS)
3 LDFLAGS += $(SANFLAGS)
4 main:  main.o l.o
5     $(CXX) -o $@ $^ $(LDFLAGS)
6 main.o: main.cc l.h
7 l.o:   l.cc l.h
```

- It *is* possible to include dependencies manually
  - See the last 2 lines of this Makefile
  - Note we don't need to provide an action for those rules, as `make` already knows how to build `.o` files
- However, that's tedious and error-prone

Automated header file dependencies


```
1 # ....
2 main:  main.o l.o
3     $(CXX) -o $@ $^ $(LDFLAGS)
4 include $(patsubst %.cc,%.dep,$(wildcard *.cc))
5 %.dep: %.cc
6     $(CXX) -MM $^ >$@
```

- We can do it automatically, though
- Passing the `-MM` option to our C++ compiler will tell it to generate a dependency line for our `Makefile`

Concluding Makefiles

- `make` is a useful tool (indispensible for larger projects) for compiling
- I will expect you to maintain your own `Makefiles` for your future assignments
  - You should not have to change the flags/compiler options
  - You will probably just have to write the name of the executable and the dependencies
  - Feel free to copy-paste the rest from what I have given you

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 `.cc` source files
- We can use `make` to automate the compiling process