CSCI 1300
Intro to Computing

Gabe Johnson

Lecture 31 April 3, 2013

C++ Compilers
Intro to C++ Syntax
Data Types

Lecture Goals

1. Test Results
2. C vs C++ and Compilers
3. Composition of C++ Programs
4. Basic Syntax and Data Types
5. If, While, For
6. Coding Session

Upcoming Homework Assignment

HW #7 **Due: Friday, Apr 3**

C++ Basics

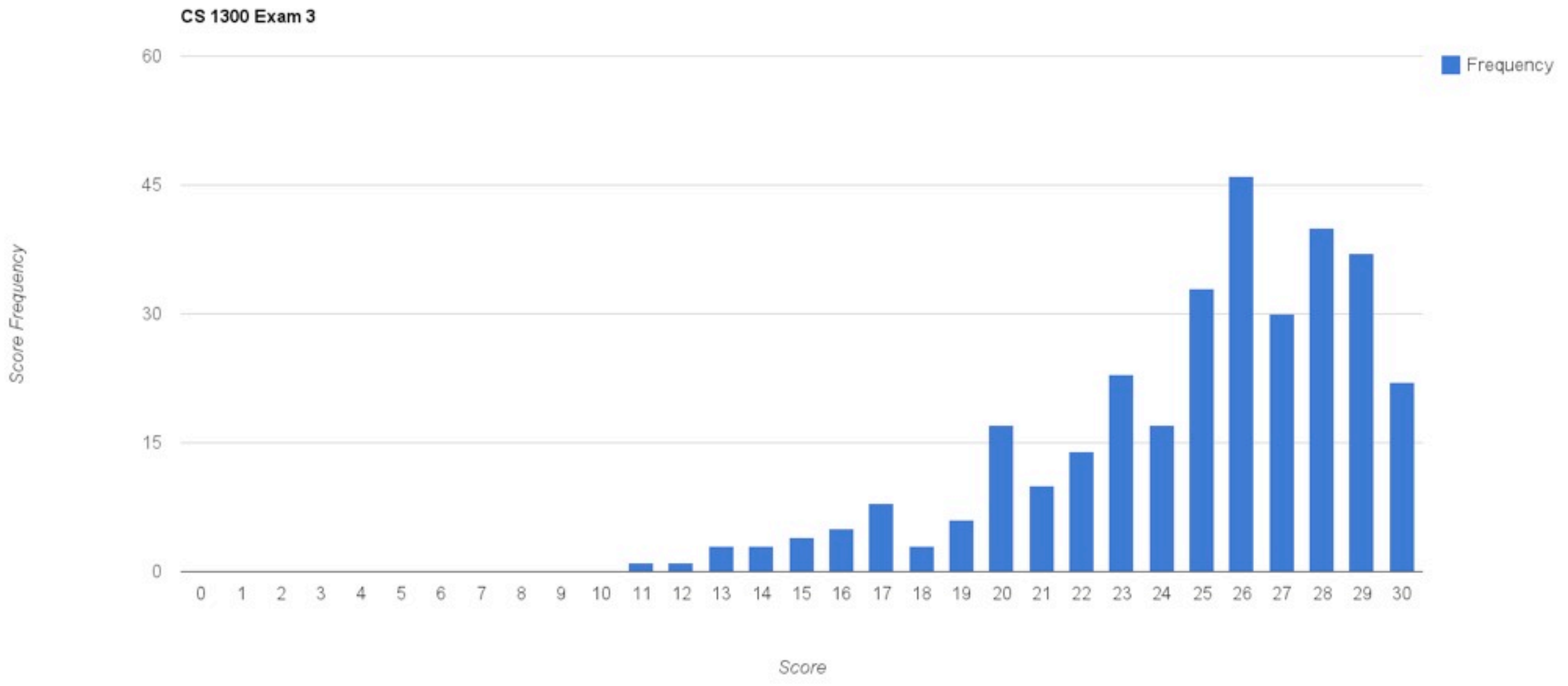
This assignment is sort of like the Basic Functions assignment (HW 2), but it is only in C++ and is slightly trickier. You will want to get started on this **NOW**. Don't wait until Friday to look at this. You will have to do some legwork outside of class, because I can't cover everything that is required to do the homework.

Project People: Turn in a summary like we talked about last time. It is not yet working on RG. I'll tweet it when it accepts project summaries.

Exam 3 Results

Max Points: 30
Mean: 24.7
S.D.: 4.4

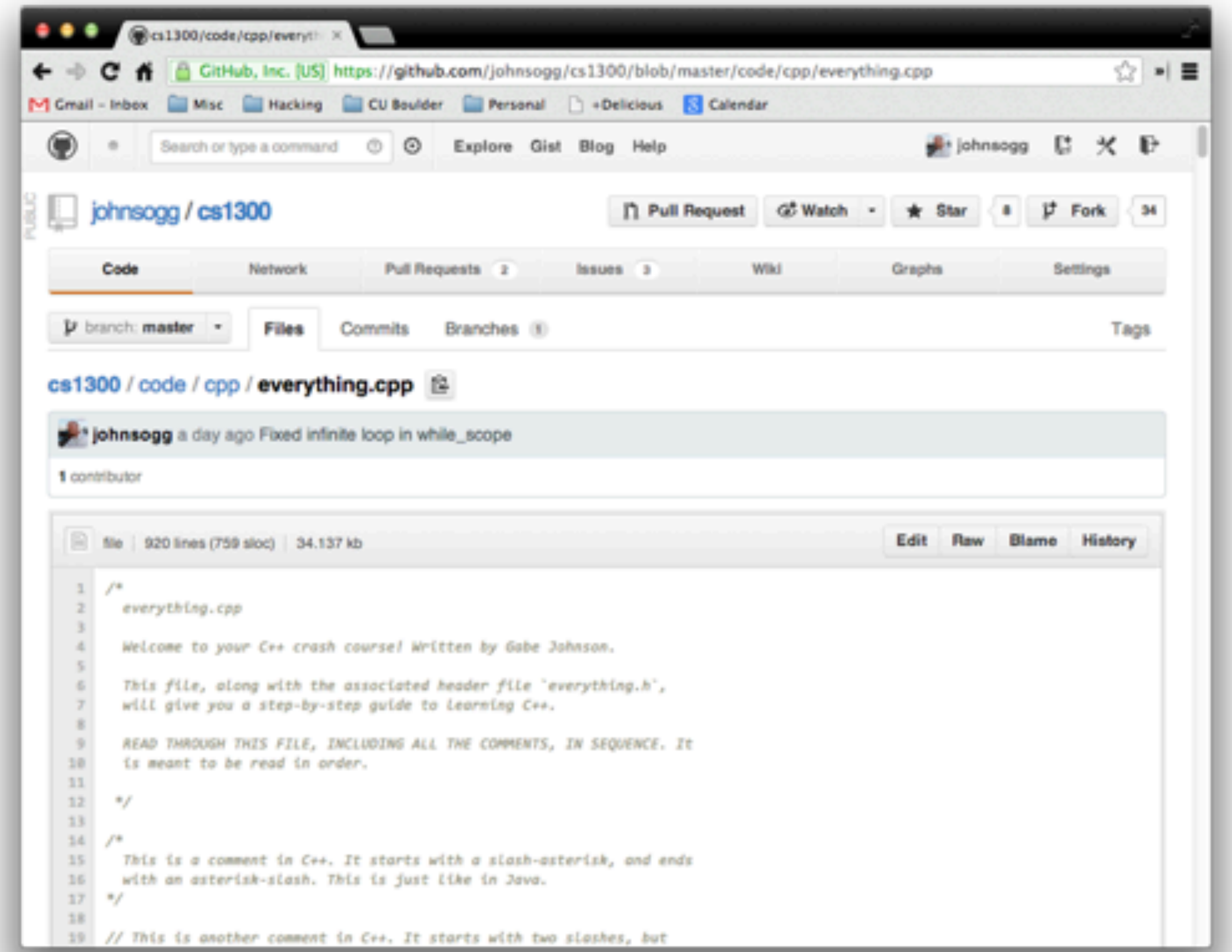
If you didn't get your test back, go to the CS office (7th floor ECOT) and bring your ID bc you'll get carded.



everything.cpp

In case you missed it:

I wrote a relatively short but concise (and hopefully good) C++ tutorial. Look for it on GitHub in:



```
/*
  everything.cpp

  Welcome to your C++ crash course! Written by Gabe Johnson.

  This file, along with the associated header file `everything.h`,
  will give you a step-by-step guide to learning C++.

  READ THROUGH THIS FILE, INCLUDING ALL THE COMMENTS, IN SEQUENCE. It
  is meant to be read in order.

 */

/*
  This is a comment in C++. It starts with a slash-asterisk, and ends
  with an asterisk-slash. This is just like in Java.
*/

// This is another comment in C++. It starts with two slashes, but
```

cs1300/code/cpp/everything.cpp

C vs C++

C: First released in the early 1970s, though it was in development since the late 60s.

C++: First released in the mid 80s.

It *adds* to C. It adds classes, and other helpful things like templates, exceptions, namespaces, and a large library called the *standard template library (STL).*

C vs C++

You can write a plain C program and compile it with a C++ compiler. This should work in all but the strangest situations.

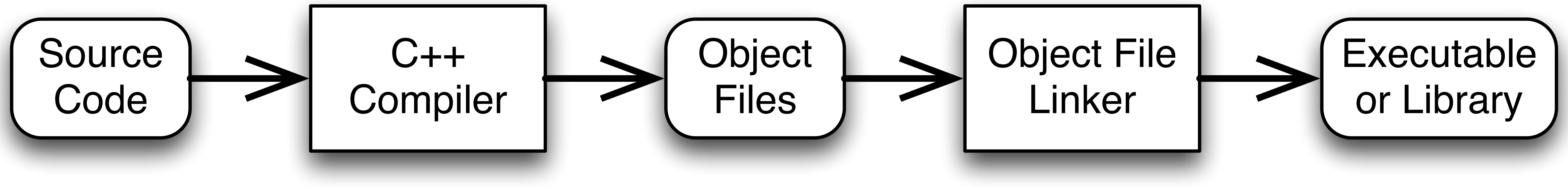
It doesn’t work the other way around, though. You can’t compile a C++ program with a C compiler.

C++ Compilers

There are several C++ compilers out there. The one on the VM is the GNU compiler. I use the one that comes with Apple's developer tools which is based on the same thing, though it has some differences that aren't really important for CS 1300.

C++ Compilers

We write *source code* in C++ (or C, if you want). This is what we spend most of our time looking at. Filenames are like `my_thing.cpp` or `my_thing.h`.



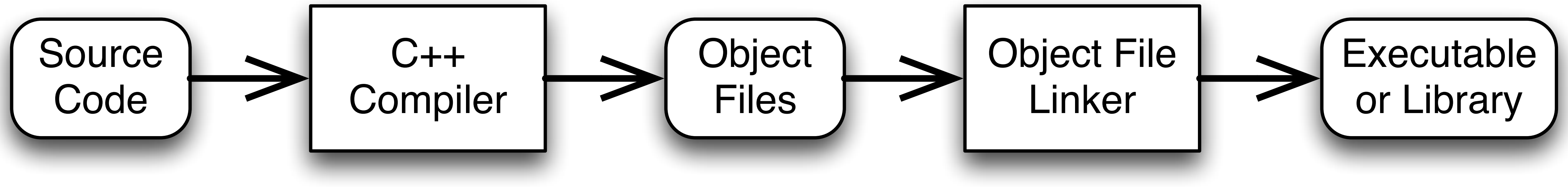
my_thing.cpp
main.cpp
my_unit_test.cpp
my_thing.h

my_thing.o
main.o
my_unit_test.o

my_thing

C++ Compilers

The compiler takes this human-readable source code and (assuming it is syntactically correct) converts it to an intermediate form called *object code*. These can be named anything but they conventionally end in `.o`.



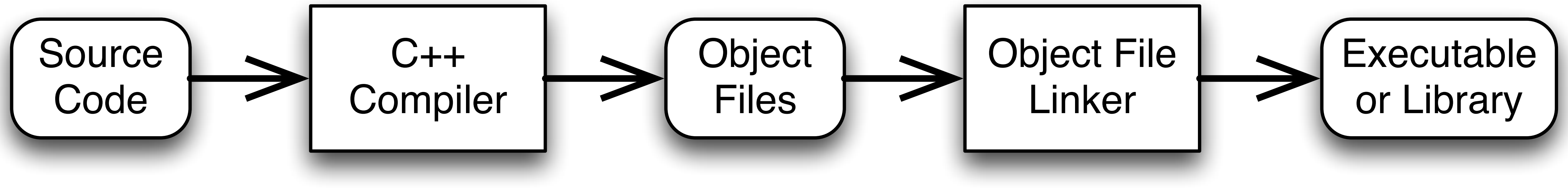
my_thing.cpp
main.cpp
my_unit_test.cpp
my_thing.h

my_thing.o
main.o
my_unit_test.o

my_thing

C++ Compilers

Next we use a program called a *linker* to take all the object files and combine them into something that we can use. Usually this is an executable program, but it could be a library. We'll only make executables here.



my_thing.cpp
main.cpp
my_unit_test.cpp
my_thing.h

my_thing.o
main.o
my_unit_test.o

my_thing

C++ Source Code

Object Code is Machine Code

When we compile a source file to an object file, we are creating machine code. This is different from Java, where we created *byte code* that is readable by a Java Virtual Machine.

In C and C++, we generate machine code that is directly usable by your computer’s processor. These are the instructions that make your CPU dance.

Object Code

You can use the program **`otool`** to examine object files. This is way beyond the scope of CS 1300, but it might help you understand what compilers do if you look directly at the output.

```
$ otool -vt everything.o
everything.o:
(__TEXT,__text) section
_main:
0000000000000000    pushq   %rbp
0000000000000001    movq    %rsp,%rbp
0000000000000004    subq    $0x10,%rsp
0000000000000008    movq    0x00000000(%rip),%rax
000000000000000f    leaq    (%rax),%rax
0000000000000012    leaq    0x00000000(%rip),%rcx
0000000000000019    movq    %rax,%rdi
000000000000001c    movq    %rcx,%rsi
000000000000001f    callq   0x00000024
0000000000000024    movq    0x00000000(%rip),%rcx
000000000000002b    leaq    (%rcx),%rcx
... (much more down below) ...
```

C++ Program Structure

C and C++ programs are broken into two kinds of files:

* Header Files: have declarations for variables, functions, and classes. They usually end in .h or .hpp.

* Implementation files: have implementations for things declared in header files. They usually end in .cpp, .cc, or .C.

C++ Program Structure

We combine these two kinds of files by using some arcane 1970s magic called preprocessor directives.

Say we are writing my_thing.cpp and we have our function declarations in my_thing.h. Near the top of my_thing.cpp we will write the following:

```
#include “my_thing.h”
```

This is as though you typed the entire contents of my_thing.h right there in my_thing.cpp.

C++ Program Structure

You can see what I mean by looking at everything.cpp, which I've mentioned a few times.

(From now on, I'll just assume that you either know about everything.cpp, or you're really not paying attention.)

Basic Syntax

Syntax in C and C++ is similar to what you've already seen in Java.

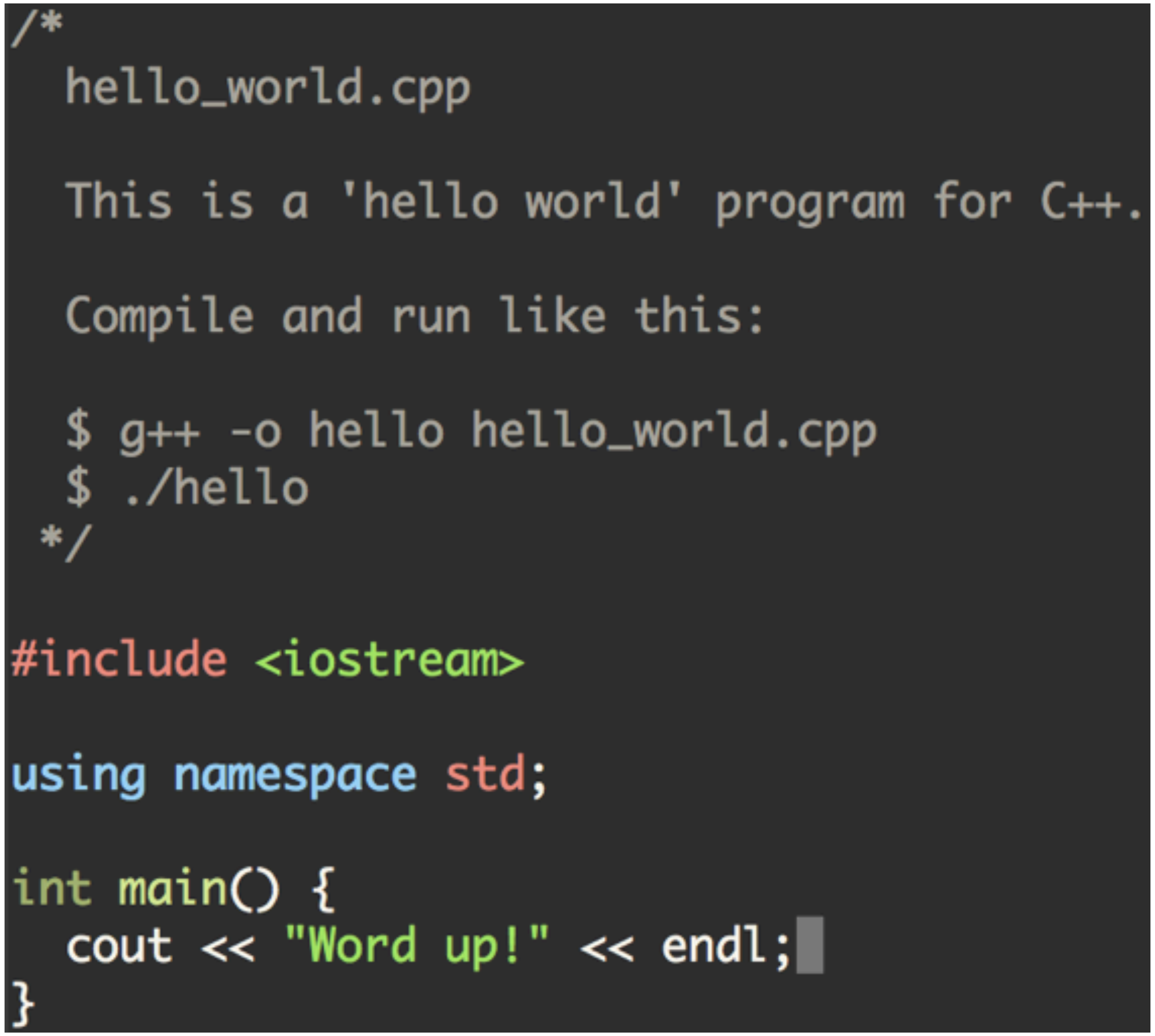
You have to tell the compiler what *type* of thing everything is, just like in Java.

It uses curly braces to group together blocks of code.

It uses semicolons to end statements.

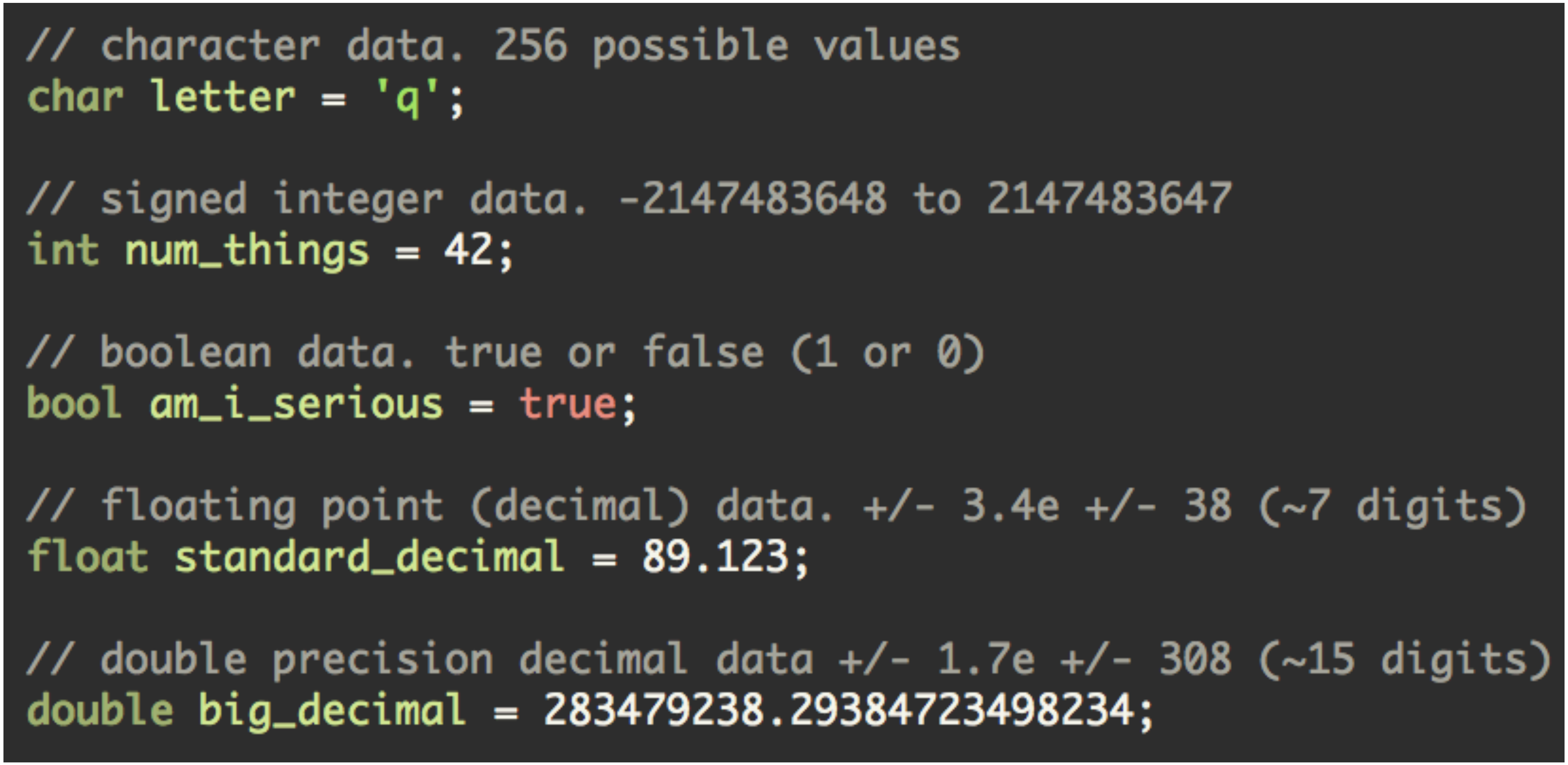
Basic Syntax

Unlike Java, C++ doesn't require you to do everything inside a class. In C++, classes work quite differently, and we won't get to that until much later (if at all).



```
/*
  hello_world.cpp

  This is a 'hello world' program for C++.

  Compile and run like this:

  $ g++ -o hello hello_world.cpp
  $ ./hello
 */

#include <iostream>

using namespace std;

int main() {
  cout << "Word up!" << endl;
}
```

Data Types (not all of them)



```cpp
// character data. 256 possible values
char letter = 'q';

// signed integer data. -2147483648 to 2147483647
int num_things = 42;

// boolean data. true or false (1 or 0)
bool am_i_serious = true;

// floating point (decimal) data. +/- 3.4e +/- 38 (~7 digits)
float standard_decimal = 89.123;

// double precision decimal data +/- 1.7e +/- 308 (~15 digits)
double big_decimal = 283479238.29384723498234;
```

Declare Variables

You have to tell the compiler what *type* of thing a variable is before you can use it. Do this by *declaring* it.

```
int num_kids;
float height;
bool happy;
```

Declarations involve two things: (1) the data type, like int or bool, and (2) the name of the variable, like num_kids or happy.

Initialize Variables

When you declare a variable, you can’t depend on what value it contains. Declaring variables just grab some spot in memory and sticks a flagpole there saying “this is mine!”. If you want to set the value contained in that memory, you have to initialize it.

Initialize Variables

You can initialize a variable when you declare it:

```
int num_kids = 2;
```

Or you can initialize it right afterwards:

```
int num_kids; // uninitialized
num_kids = 2; // initialized
```

Function Declarations

We saw a function declaration earlier in the hello world screenshot. You can make your own function like this:

```
int add_numbers(int first, int second) {
  int sum = first + second;
  return sum;
}
```

Function Syntax

```
int add_numbers(int first, int second) {
  int sum = first + second;
  return sum;
}
```

Function returns an int.

Function is called ‘add_numbers’.

Function takes two parameters:

- an int called ‘first’
- an int called ‘second’

Function has a block of code to execute when called.

Calling Functions

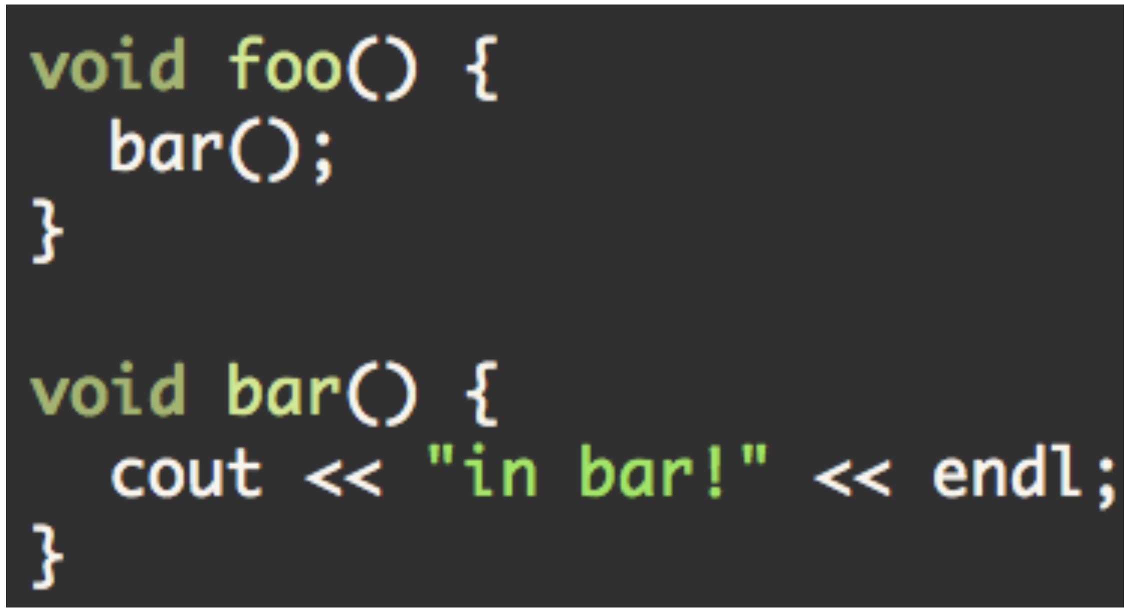
We can then call a function in much the same way we did in Python and Java.

```
int result = add_numbers(10, 4);
```

Notice that ‘result’ has to be an integer since that is what add_numbers returns.

Unknown Functions?

Like Python, C++ has to know about a function before we can call it. We don't have to *define* it, we just have to *declare* it. E.g. this code won't work:



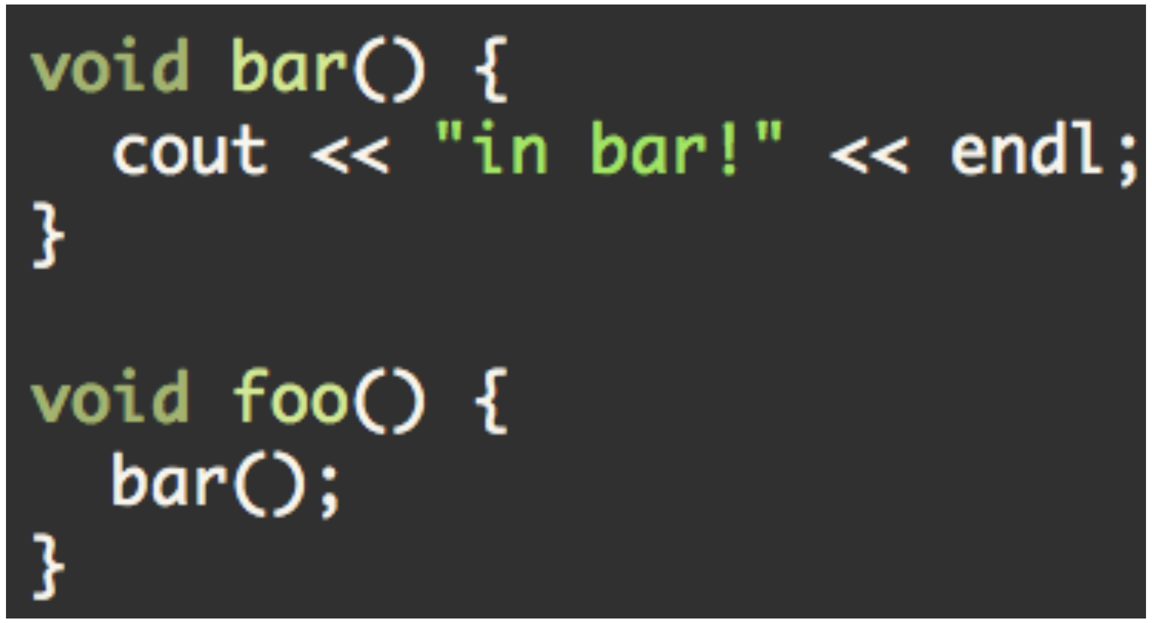
```
void foo() {
  bar();
}

void bar() {
  cout << "in bar!" << endl;
}
```

The compiler will give the following error:

```
lec31.cpp: In function 'void foo()':
lec31.cpp:40: error: 'bar' was not declared in this scope
```

Unknown Functions?

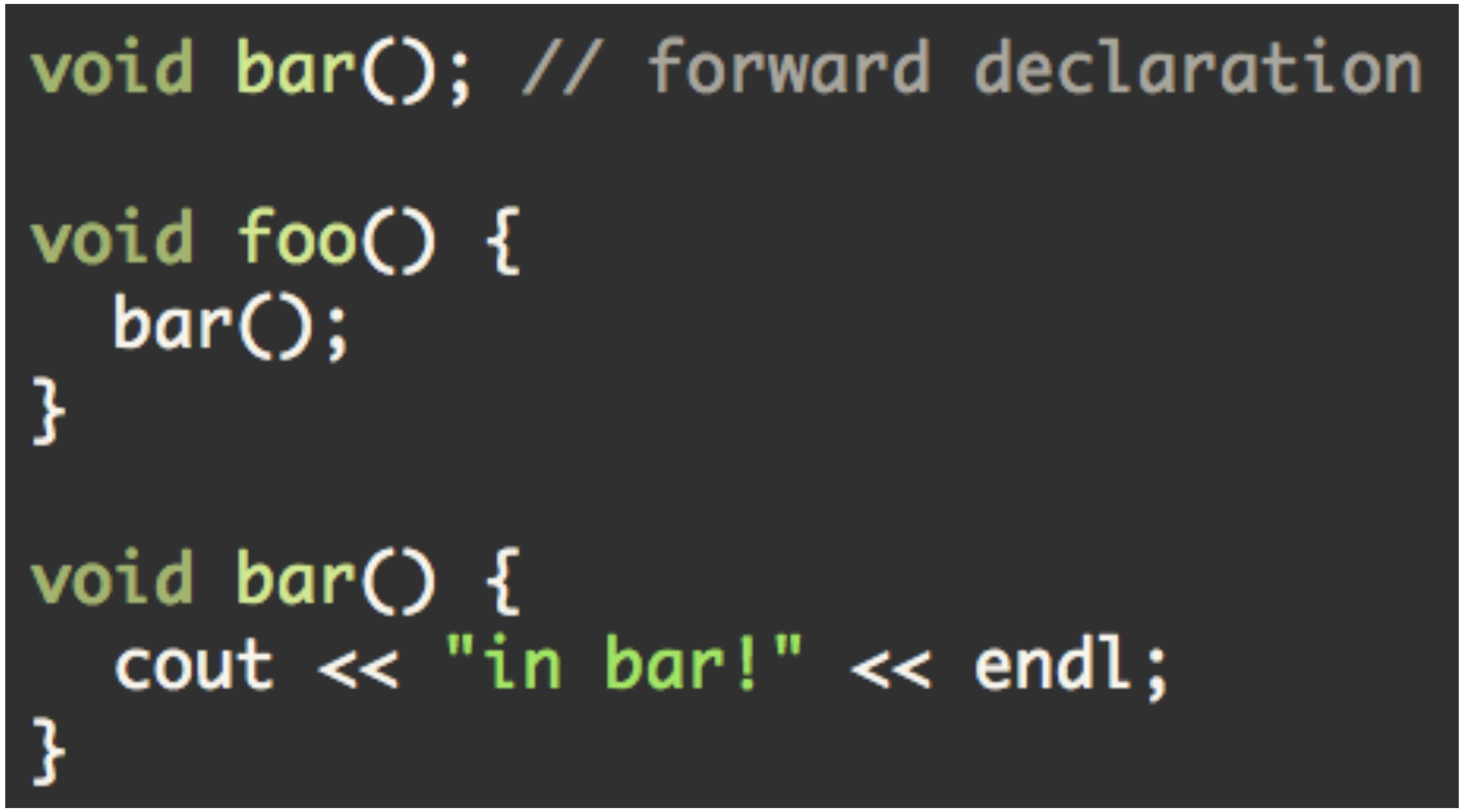
One way to fix this is to ensure that ‘bar’ is defined above ‘foo’.



```
void bar() {
  cout << "in bar!" << endl;
}

void foo() {
  bar();
}
```

Unknown Functions?

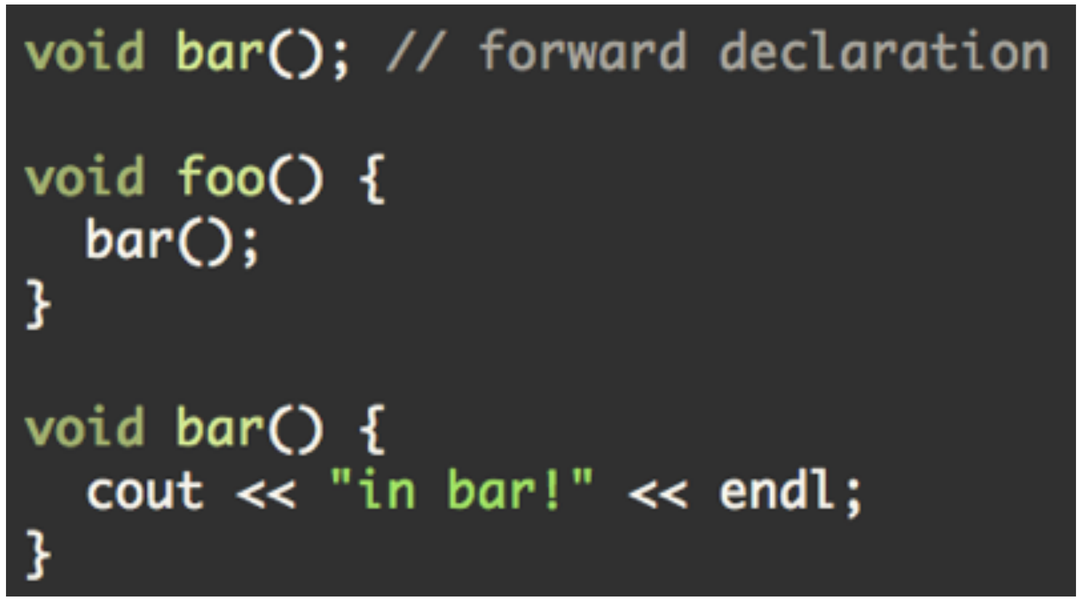
Another way to fix this is to simply declare the ‘foo’ function before it is referenced. We don’t need to actually *define* it. We only need to *declare* it.



```
void bar(); // forward declaration

void foo() {
  bar();
}

void bar() {
  cout << "in bar!" << endl;
}
```

That’s what .h is for

This last thing we did is why we use header files. It is convenient to declare all of your functions at once, at the top of the file, so you don’t have to worry about what order things appear in your implementation. So we put them all in a header file and then #include it.



```cpp
void bar(); // forward declaration

void foo() {
  bar();
}

void bar() {
  cout << "in bar!" << endl;
}
```

If Statements

General format is:

```
if (condition) {
  // code
}
```

The 'else' clause must come last, if you have it at all. 'else if' clauses must come in between, if you have them at all.

Could also have else if and else clauses:

```
if (cond1) {
  // code
} else if (cond2) {
  // code
} else if (cond3) {
  // code
} else {
  // default code
}
```

While Loops

General Format:

```
while (condition) {
  // code
}
```

This is basically identical to how Java does it. I can't think of any differences.

For Loops

General Format:

```
for (int i=0; i < some_number; i++) {
  // code
}
```

There’s three ‘slots’ inside those parenthesis, just like in Java, and they behave the same too.

For Loops

General Format:

```
for (int i=0; i < some_number; i++) {
  // code
}
```

The first slot (int i=0) is run one time before the first time through the code. It is almost always used to initialize a counting variable. That variable is almost always declared there, as well.

For Loops

General Format:

```
for (int i=0; i < some_number; i++) {
  // code
}
```

The next slot (i < some_number) is run at the beginning of every loop. It should be a boolean expression (it gives true/false), and the code only runs if that expression is true.

For Loops

General Format:

```
for (int i=0; i < some_number; i++) {
  // code
}
```

The last slot (i++) is run at the end of every loop. It usually increments the variable that was declared in the first slot.