Just Enough C++

44-440/640-IoT

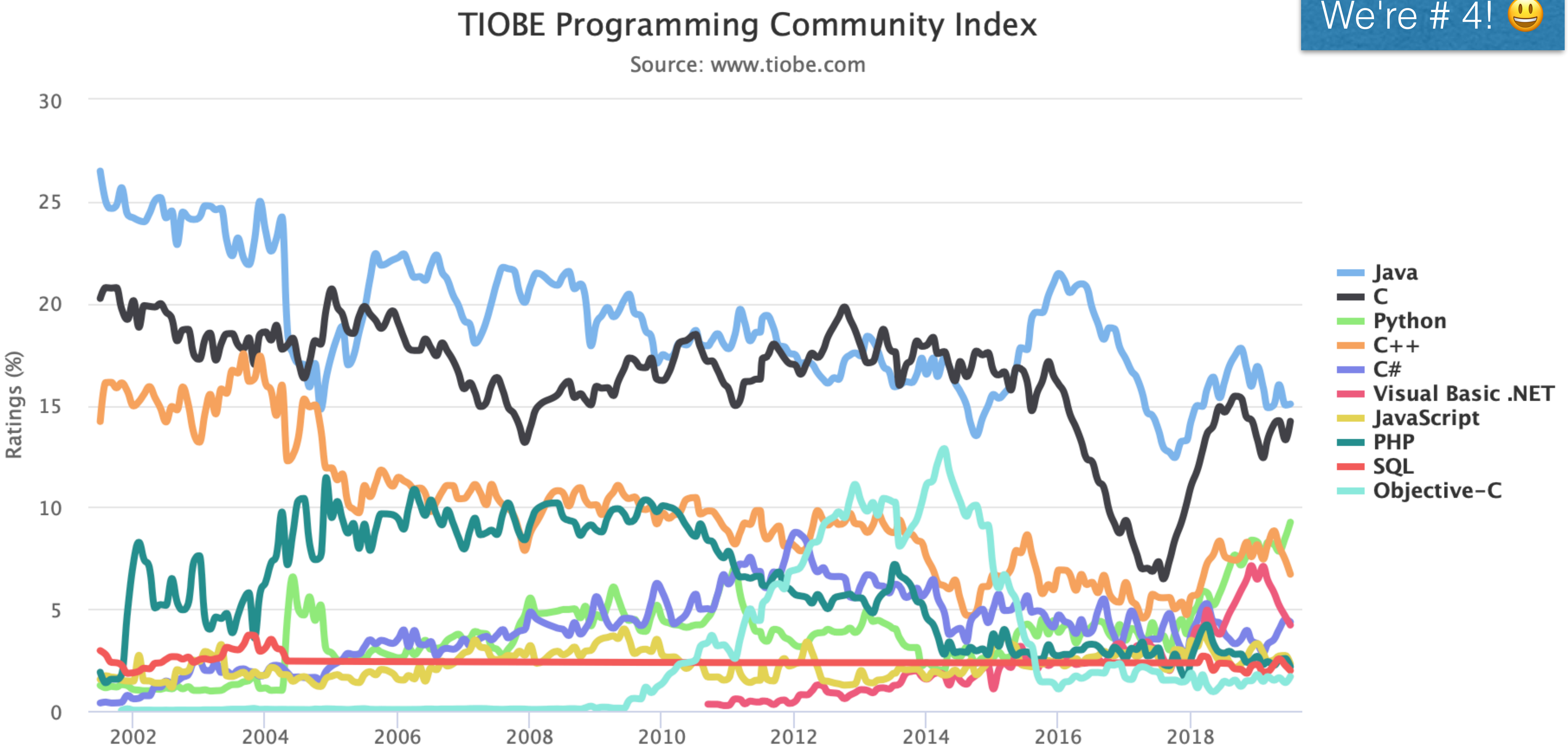
During class we will concentrate on slides 6, 8, 15, 16, 21, 24: read through the rest prior to doing

Objectives

- Students will be able to
    - write a basic "hello world" program in C++
    - read input, write output using streams
    - enumerate the primitive types found in C++
    - construct programs using decision structures and loops
    - write functions
    - declare arrays

C++ In the Scheme of Things

We're # 4! 😃
We're # 4! 😃



Motivation for Learning C++

- The Photon can be programmed using **Wiring**, basically a framework that sits on top of C++ and provides extra functionality.

- Unlike C++, Wiring programs do not use main(), and allow forward references to variables/functions/classes: however, most C++ code will work just fine

- In order to program the Photon, therefore, it behooves us to take a *quick* glance at C++, which we will do here, deferring coverage of Wiring-specific functions for another day.

Overview of C++

- Created by Bjarne Stroustrup, C++ was originally dubbed "C with Classes", because that's what it did: add OOP functionality to C. C++ is a big, complex language*: of which we only need a subset.

- Syntactically, C++, C and Java are *quite* similar. When in doubt, C/Java syntax will likely work: variable declarations, loops, decision structures, function declarations, etc., all look the same in C++ as in C and Java.

- There are some differences:

	Java	C++
Source code	translated to byte code that runs on a virtual machine	compiled to native code for a particular CPU
Classes	everything must be in a class	can forget about classes and objects when writing C++ (although they do have their place!)
Pointers	does not use them explicitly	loves them!
Passing Arguments	Pass by value, always	Can pass by value (default) or by reference (using &)
Garbage Collection	Built-in and automatic	You are entirely responsible for memory management. Good luck.

**sometimes disparaged for being so:*

Output: Hello, World!

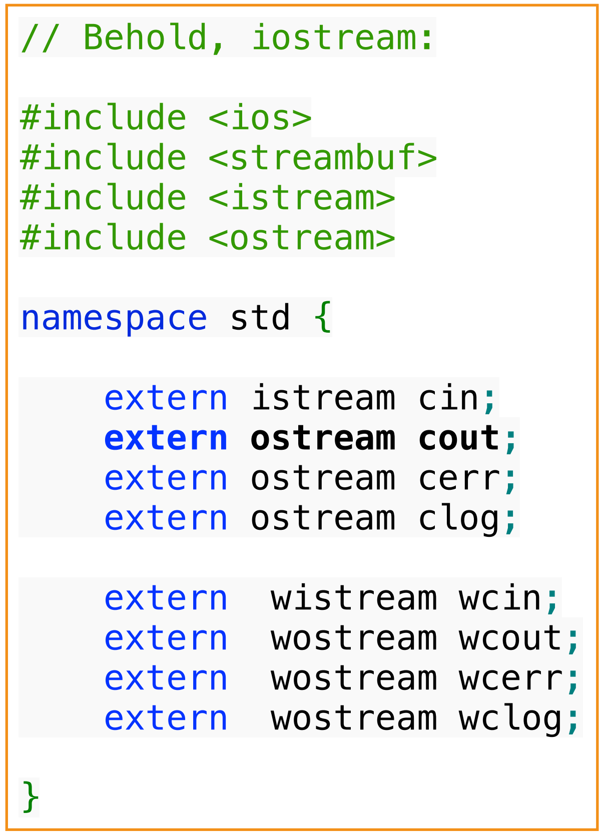
```cpp
#include <iostream>
int main(){
  std::cout << "Hello, World!\n";
  return 0;
}
```

Hello, World! Dissected

- A *complete* C++ program requires a function, **main()**, that returns an int to the OS.

- **#include <iostream>** is a preprocessor statement that copies the contents of iostream into our program, so that the compiler will know that **std::cout** is an ostream (output stream) object.

- **std::cout** is connected to standard output: anything sent to it with the **<<** operator, will end up displayed to the user.



```
// Behold, iostream:

#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {

    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;

    extern  wistream wcin;
    extern  wostream wcout;
    extern  wostream wcerr;
    extern  wostream wclog;

}
```

```
#include <iostream> // Now we can do output using the cout object

int main(){
  std::cout << "Hello, World!\n";
  return 0;
}
```

Hello, Variables!

```
#include <iostream>

int main() {

    double width;
    double height;
    double area;

    std::cout << "Enter width, height: ";
    std::cin >> width >> height;

    area = width * height;

    std::cout << "A " << width << " x " << height << " rectangle has an area of " << area << std::endl;
    printf("A %f x %f rectangle has an area of %f\n", width, height, area);
    return 0;
}

/*
 Output:
 Enter width, height: 25 35
 A 25 x 35 rectangle has an area of 875
 */
```

Hello, Variables! Dissected

- Variables are declared with *type variableName;* statements

- **std::cin** is connected to standard input: we read from it using the **>>** operator. We can chain >> operators together to read multiple values

- The << operator can be chained together to send multiple items to standard output

- **std::endl** is the end of line character (either '\n' or '\r''\n', depending on the OS)

- Everything else is syntactically identical to C or Java.

Primitive Types

- **char** (in single quotes)
- **bool** (true or false)
- **float**, **double**, **long double** (all use IEEE 754, and usually 32, 64 and 80 bits)
- **short**, **int**, **long**, **long long** (really)
- As in C, chars and ints can be signed (the default) or unsigned (which gets you an extra bit (literally, 1 bit) of data, which in theory can be useful if the values you work with are always non-negative. In practice, unsigned ints tend to cause grief: just use a bigger signed type (long or long long)
- C++ does not, by itself, specify the precise size of the primitive types, but it does guarantee that a char is 1 byte, and the following ordering:

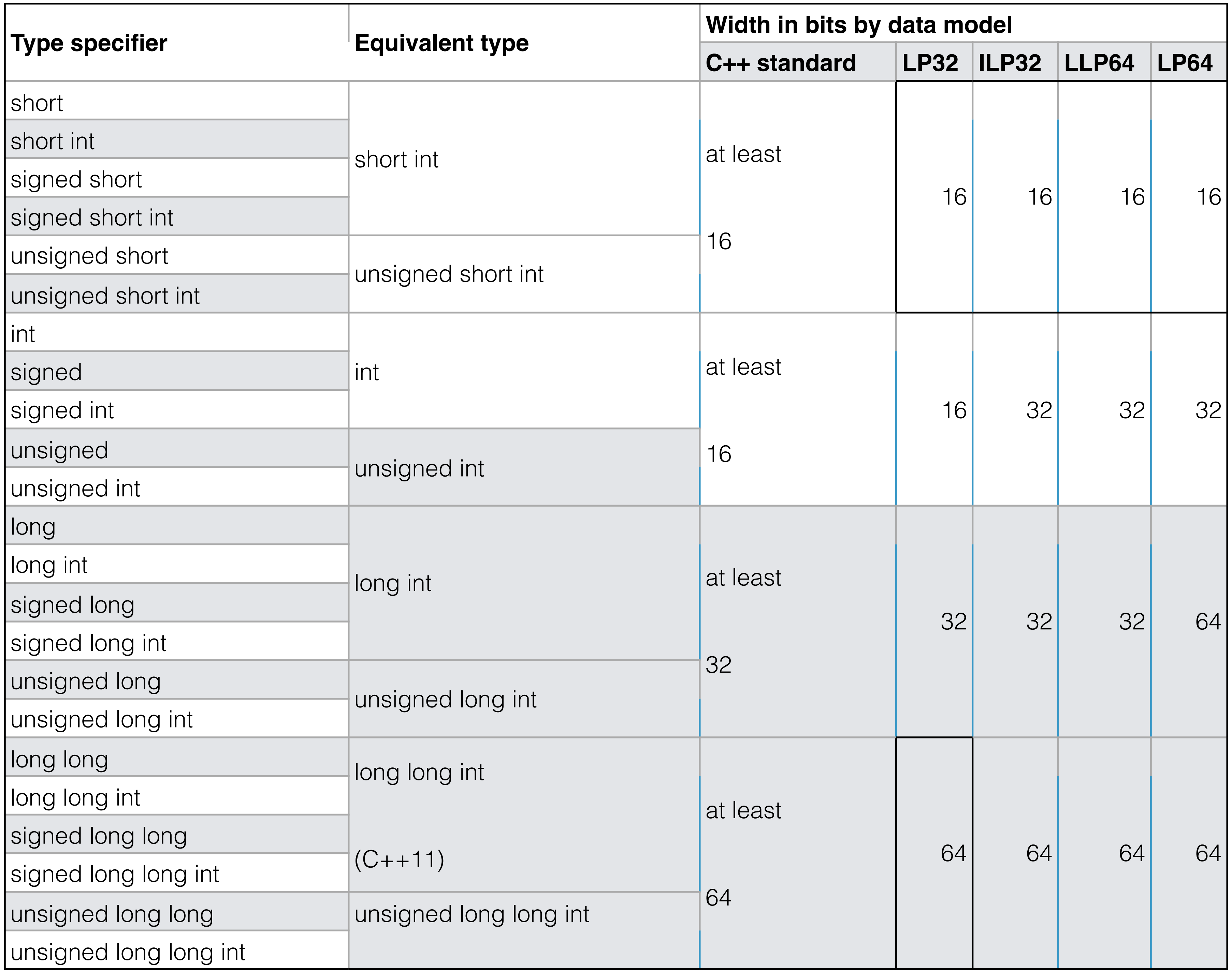
```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

Techy Aside*: Why Unsigned Ints are a Bad Idea

```cpp
int main()
{
    unsigned int x = 3;
    unsigned int y = 5;

    std::cout << "Behold! Yes, 3 - 5 = " << x - y << std::endl;
    return 0;
}

// Behold! Yes, 3 - 5 = 4294967294

/*
  3 - 5
= 0x00 00 00 03 - 0x00 00 00 05
= 0x00 00 00 03 + FF FF FF FB
= 0xFF FF FF FE
= 15*16^7 + 15*16^6 + 15*16^5 + 15*16^4 + 15*16^3 + 15*16^2 + 15*16^1 + 14
= 4294967294
*/
```

*Techy Asides are for reference and will not be on the exam, but your instructor will be pleased to answer any questions about them 😉

Techy Aside: A Plethora of ints



Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
short	short int	at least 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int					
unsigned long long int						

Techy Aside: Operator Precedence

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() type{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
9	!=	For relational operators = and ≠ respectively	
10	a&b	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
16	,	Comma	Left-to-right

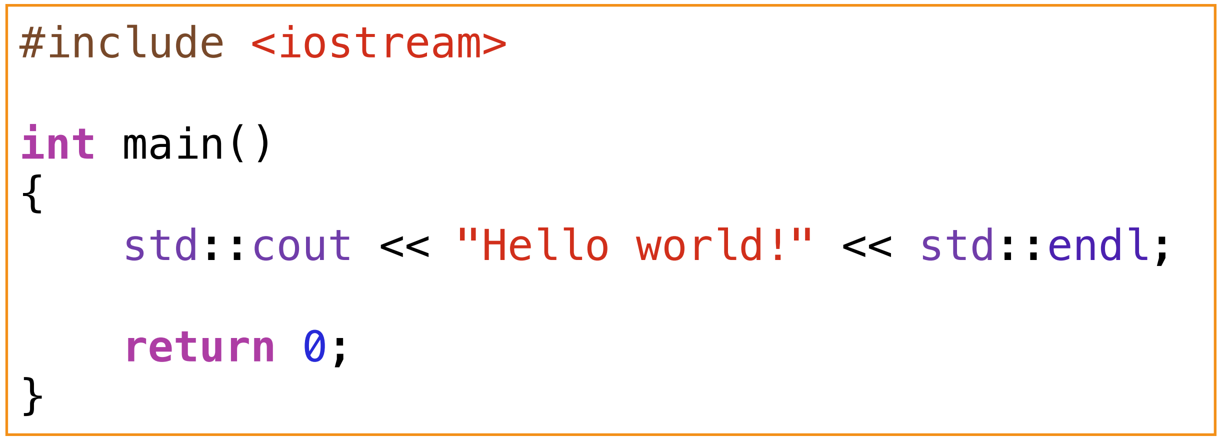
Namespaces

- **namespaces** are collections of identifiers. They are akin to Java packages.

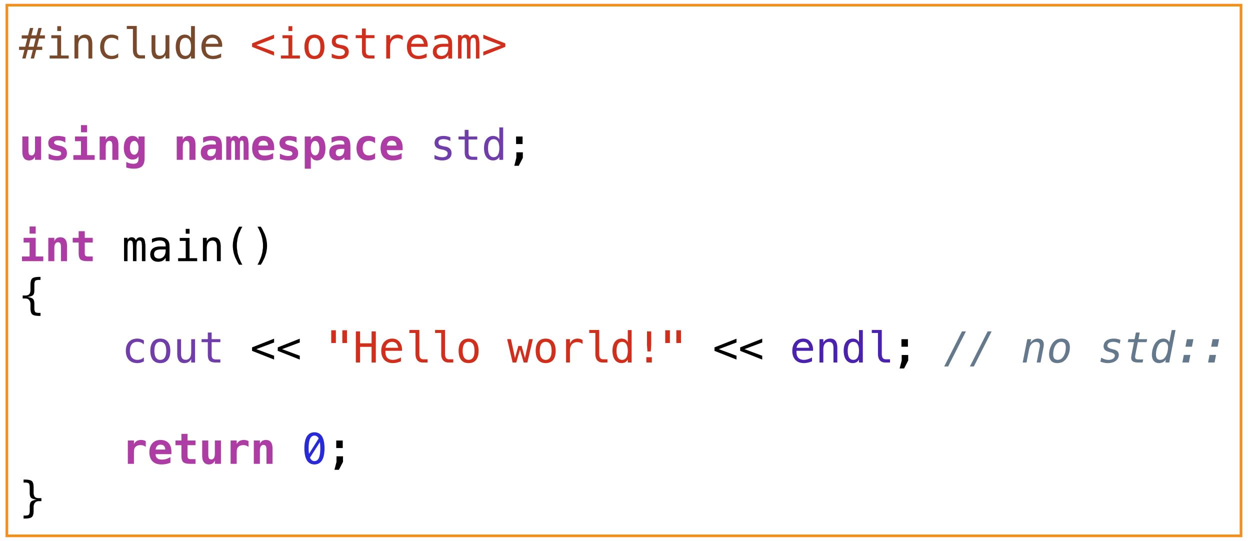
- The most common namespace, std, includes such identifiers as cin, cout, endl, vector<t>, string, etc.

- To refer to a namespace identifier *without* having to write std::, write **using namespace std;**

- This is considered poor form by some authors, but you will see it in practice.



```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;

    return 0;
}
```



```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl; // no std::

    return 0;
}
```

Fun with Functions

```
#include <iostream>
int sum(int a, int b, int c) {
    return a + b + c;
}


int main(){

    int x;
    int y;
    int z;

    std::cout << "Enter 3 numbers to sum: ";
    std::cin >> x >> y >> z;

    std::cout << "Their sum is " << sum(x,y,z) << std::endl;
    return 0;
}
```

The compiler must have *seen* a function, or at least its prototype, before that function is invoked

Here, the compiler runs into the sum() function first, so it will be content

Fun with Functions

```
#include <iostream>

int main(){

    int x;
    int y;
    int z;

    std::cout << "Enter 3 numbers to sum: ";
    std::cin >> x >> y >> z;

    std::cout << "Their sum is " << sum(x,y,z) << std::endl; // Use of undeclared identifier 'sum'
    return 0;
}

int sum(int a, int b, int c) {
    return a + b + c;
}
```

Trouble in River City! sum() is invoked before it is defined

Fun with Functions

```
#include <iostream>

int sum(int a, int b, int c);

int main(){

    int x;
    int y;
    int z;

    std::cout << "Enter 3 numbers to sum: ";
    std::cin >> x >> y >> z;

    std::cout << "Their sum is " << sum(x,y,z) << std::endl; // No probs
    return 0;
}

int sum(int a, int b, int c) {
    return a + b + c;
}
```

Here, the prototype is seen before the invocation, so we are good.

Organizational Strategy

- Place function prototypes in a .h file
- Place function definitions in a .cpp file
- Any other file that needs to use the functions should **#include** the .h file
- #include inserts the contents of the file, so the compiler really does see the function prototypes and can check that the arguments are correct

Organizational Strategy

functionfun.h

```
#ifndef functionfun_h
#define functionfun_h

int factorial(int n);
int fibonacci(int n);

#endif
```

functionfun.cpp

```
#include "functionfun.h"

int factorial(int n){
    if(n == 0 || n == 1){
        return 1;
    }

    return n * factorial(n-1);
}

int fibonacci(int n){
    if(n == 0 || n == 1){
        return 1;
    }

    return fibonacci(n-1) + fibonacci(n-2);
}
```

```
#include <iostream>
#include "functionfun.h"

int main(){

    for(int i=0; i < 50; i++){
        std::cout << i << " -- " << factorial(i) << std::endl;
    }

    return 0;
}
```

What the programmer writes

Organizational Strategy

functionfun.h

```
#ifndef functionfun_h
#define functionfun_h

int factorial(int n);
int fibonacci(int n);

#endif
```

```
#include <iostream>
int factorial(int n);
int fibonacci(int n);

int main(){

    for(int i=0; i < 50; i++){
        std::cout << i << " -- " << factorial(i) << std::endl;
    }

    return 0;
}
```

functionfun.cpp

```
#include "functionfun_h"

int factorial(int n){
    if(n == 0 || n == 1){
        return 1;
    }

    return n * factorial(n-1);
}

int fibonacci(int n){
    if(n == 0 || n == 1){
        return 1;
    }

    return fibonacci(n-1) + fibonacci(n-2);
}
```

What the compiler sees

Arrays

```
#include <iostream>

int timing[5] = {10, 30, 20, 5, 18};
char names[3] = {'a', 'b', 'c'};
//double wontwork[]; // wontwork won't work -- must be an explicit size in [ ], or an initializer
double willWork[] = {3.5, 10.2, 18.6}; // no explicit size, so it will be 3 elements long

int main(){

    for(int i=0; i < 5; i++){
        cout << timing[i] << '\n';
    }

    cout << "We are done" << endl;
}
```

Dynamically Sized Arrays

- This will require an understanding of pointers ... so let's learn about pointers!

Global v. Local Variables

- **Local** variables, declared in a block { }, have **block scope** (visible only in their block)

- **Global** variables, declared outside of any block, have **file scope** (visible throughout the file)

- Local variables are (often) declared at the top of their block; Global variables are (usually) declared at the top of their file

- Local variables, by default, have **automatic duration** - created when the block is entered, destroyed when the block is exited

- Global variables have **static duration** - created when the program starts, destroyed when it ends

Global & Local Demo

```cpp
#include <iostream>
#include "library.h"

extern int numRandyCalls; // global, defined externally

int main()
{
    int i = 42; // local to main() - automatic duration

    std::cout << "Global i: " << i << std::endl;

    for(int i=0; i < 10; i++){  // local to the for-loop - automatic duration
        std::cout << i << ". " << randy() % 10 << std::endl;
    }

    std::cout << "randy() was invoked " << numRandyCalls << " times." << std::endl;

    return 0;
}
```

```cpp
int numRandyCalls = 0; // global variable, declared here - it has static duration
static bool notVisibleOutside = false;

// returns pseudo random numbers between 0 and m-1
unsigned int randy(){
    const int a = 48271;
    const int m = 2147483647;
    static int x = 10; // static here means that x's value is preserved, over repeated calls


    x = a*x % m;
    numRandyCalls++;

    return x;
}
```

Global & Local Demo, Dissected

- numRandyCalls is **global**, outside of any block

- **extern** indicates that the variable is defined elsewhere

- **static** indicates that notVisibleOutside, although global, cannot be used outside of its file (it will crash with a link error)

- There are two versions of i: the innermost version masks the other

Techy Aside: Initialization

- C++ provides a *plethora* of techniques for initializing a variable. In the following, type is any valid C++ type:
  - type variable = value; // C-like initialization
  - type variable (value); // constructor initialization
  - type variable {value}; // uniform or list initialization
  - auto variable = value; // type deduction
- In this course, we will use C-like initialization unless otherwise needed

ICE

- Go to www.repl.it or www.codechef.com/ide (yay, for online IDEs 😀)
- Write a function, equiNum(), that will be passed in 3 int arguments and return the number that are equal.
  - e.g., equiNum(1,1,1) ==> 3
  - equiNum(1,1,2) ==> 2
  - equiNum(1,2,3) ==> 0
- Invoke equiNum() 3 times, printing out the results on 3 lines, using the arguments (1,1,1), (1,1,2) and (1,2,3).

Resources

- http://wiki.wiring.co/wiki/C/C%2B%2B_Comparison
- http://en.cppreference.com/w/
- http://en.cppreference.com/w/cpp/language/types
- https://community.particle.io/t/how-pure-is-the-c-in-photon/23438/4
- Olsson, Mikael. C++17 Quick Syntax Reference: A Pocket Guide to the Language, APIs and Library, Apress, 2018