

What is C++?

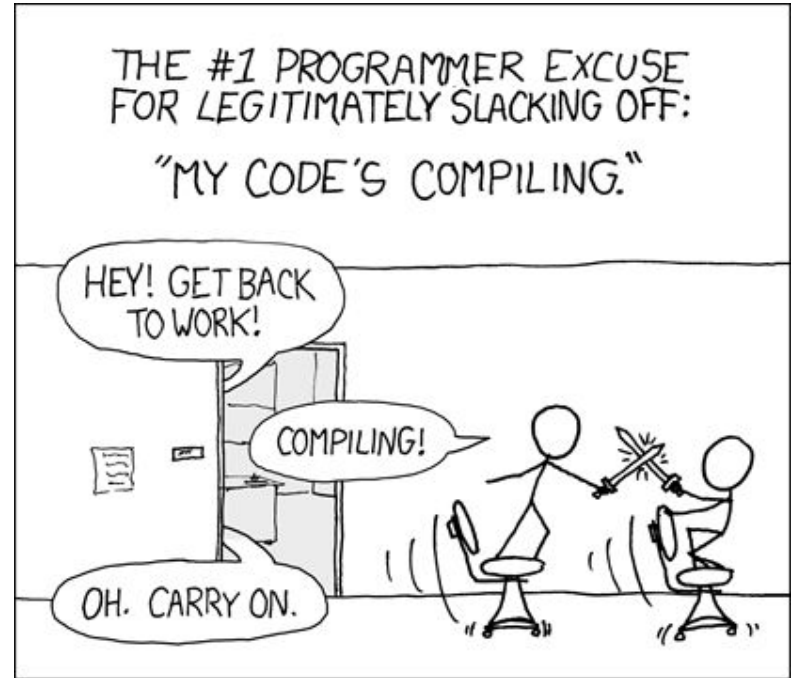
- General-purpose programming language
 - Imperative
 - Object-oriented
- Developed by Bjarne Stroustrup between 1979-83
 - Enhanced C with Simula-like features
 - C++ 2.0 released in 1989
 - C++11 released in 2011

Why C++?

- Every programmer needs to know two classes of language
 - Script-y language for everyday / simple kinds of things
 - Ex: Python, javascript
 - System-y kind of language that provides speed, efficiency, power to do harder, more computational stuff
 - Ex: C++, C

Differences between Python and C++

- Code is compiled
 - No interpreter
 - Have to do most of the bookkeeping ourselves
 - Memory creation / destruction
 - No garbage collector
- Executable created from your code



Types of error

- Compile-Time Error

- Broke the rules of C++
- Compiler doesn't know what you want to do
- Compiler flags the error and quits
- **No executable created!**

- Runtime Error

- Followed the rules of C++ but still messed up
- Executable was created but it did something wrong

Standard Template Library (STL)

- Provides libraries that handle particular tasks
 - Containers
 - Algorithms
 - Memory management

Differences between Python and C++

- Python
 - Dynamic typing (determined at run-time)
 - Duck typing (if it quacks like a duck)
- C++
 - Static typing (determined at compile-time)
 - Lots of types
 - Modifiers to those types
 - The compiler must know the types of everything before it will compile

Example

```
const map<string, vector<long>> const * m = &some_var;
```

- **Green** stuff is all part of a single type declaration
 - A constant pointer to a constant map of strings to a vector of longs
- You have to declare your variable's type
- It only holds that type (or does something weird to what you try to stuff into that type)

Some context required

- Syntax requires context
 - C++ tries to stay backwards-compatible with C (mostly)
 - C++ continues to add features to the language that really help
 - There are only so many characters on a standard keyboard
- Results in symbols that get reused to mean different things
 - The symbol is not enough to sort out what it means
 - Ex: `aa bb(cc);`
 - Declaration of object `bb` of type `aa` with variable `cc` passed to its constructor
 - Declaration of function `bb` that takes an object of type `cc` and return type `aa`

Another Example

```
long my_int = 123;  
long &ref_int = my_int;  
long *another_int = &my_int;  
my_int = * another_int;  
my_int = my_int * my_int;
```

- & means “a reference type” in line 2 and “the address of” in line 3
- * means “pointer type” in line 3, “value pointer points to” in line 3 and multiplication in line 5

Error Messages

- C++ has a well-earned reputation for cryptic error-messages
- A single syntax error can generate 10s of lines of error.
 - Most important info is the line number

```

/usr/include/c++/7/bits/stl_iterator.h:343:5: note: template argument deduction
n/substitution failed:
sizes.cpp:9:39: note: 'std::basic_ostream<char>::__ostream_type {aka std::basi
c_ostream<char>}' is not derived from 'const std::reverse_iterator<_Iterator>'
    cout << "short: " << sizeof(short) << endl;
    ~~~~~
In file included from /usr/include/c++/7/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/7/bits/char_traits.h:39,
                 from /usr/include/c++/7/ios:40,
                 from /usr/include/c++/7/ostream:38,
                 from /usr/include/c++/7/iostream:39,
                 from sizes.cpp:1:
/usr/include/c++/7/bits/stl_iterator.h:305:5: note: candidate: template<class _I
terator> bool std::operator<(const std::reverse_iterator<_Iterator>&, const std:
:reverse_iterator<_Iterator>&)
    operator<(const reverse_iterator<_Iterator>& __x,
    ~~~~~
/usr/include/c++/7/bits/stl_iterator.h:305:5: note: template argument deductio
n/substitution failed:
sizes.cpp:9:39: note: 'std::basic_ostream<char>::__ostream_type {aka std::basi
c_ostream<char>}' is not derived from 'const std::reverse_iterator<_Iterator>'
    cout << "short: " << sizeof(short) << endl;
    ~~~~~
In file included from /usr/include/c++/7/bits/stl_algobase.h:64:0,
                 from /usr/include/c++/7/bits/char_traits.h:39,
                 from /usr/include/c++/7/ios:40,
                 from /usr/include/c++/7/ostream:38,
                 from /usr/include/c++/7/iostream:39,
                 from sizes.cpp:1:
/usr/include/c++/7/bits/stl_pair.h:449:5: note: candidate: template<class _T1, c
lass _T2> constexpr bool std::operator<(const std::pair<_T1, _T2>&, const std::p
air<_T1, _T2>&)
    operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    ~~~~~
/usr/include/c++/7/bits/stl_pair.h:449:5: note: template argument deduction/su
bstitution failed:
sizes.cpp:9:39: note: 'std::basic_ostream<char>::__ostream_type {aka std::basi
c_ostream<char>}' is not derived from 'const std::pair<_T1, _T2>'
    cout << "short: " << sizeof(short) << endl;
    ~~~~~

```

```
cout << "Size of short:" << sizeof(short) < endl;
```

Compile early, compile often

- You should *compile all the time*
 - Write a line
 - Compile
 - Repeat
- If you don't you will suffer
 - Especially early when you don't understand what the error message mean

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */

int main() {
    std::cout << "Hello, World!"; // output
}
```

includes

- To use aspects of the library system, you must include the definitions into the system
 - Like import in python
 - Table A.1 (pg 866-870) lists many of the elements and their associated include file

#

- `#include<iostream>`
is a pre-processor statement
- It does *not* end in a semicolon
 - Part of the pre-processor, not C++
- What you include goes between the chevrons < and >
 - No `.h` or `.hpp` at the end of the `include`

Comments, two kinds

- `/* ... */`

- Anything between those symbols are ignored
- Multiline

- `// ...`

- Everything after this is ignored
- One line only

main

- A runnable program requires a specific function named `main`
- This function is what will start the program when the compile code is loaded

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */
```

Return type	Func name	Argument list (now empty)
----------------	--------------	------------------------------

int	main	() {
-----	------	------

Begin and end of a block

```
std::cout << "hello, world"; // output
```

}



- Curly braces have multiple meanings
 - A **block** in the context of a function or a control statement (most common)
 - A **sequence** of statements that fill in for a single statement
- Indentation (and whitespace) means nothing
 - Only helps the reader
 - Like comments, it is helpful for making your code more readable
 - Can put many statements on a single line
 - The following is valid (but sucks)

```
#include <iostream> /* hello world program*/ int main() { std::cout <<  
"hello, world" << std::endl; }
```

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */

int main() {
    std::cout << "hello, world"; // output
}
std namespace      cout
                    stream
```

std

- When you use `include` you insert definitions into your program
- Those elements are in namespaces
 - We'll talk more about those later
 - Need to reference them by their namespace
 - The common libraries are in the `std` namespace
 - Short for standard
 - Yes, it is an unfortunate acronym

The lazy typist

- No group of human beings are lazier typists than programmers
 - Short acronyms instead of full names
 - Short variable names
- Need to find a balance between readability and typing
- Different communities have different customs
 - Java tends to favor more descriptive names
 - Unix names: remove all of the vowels, randomly drop some letters, and maybe add a vowel back

Scope resolution operator

- To differentiate namespaces, you include the namespace name followed by `::`
- `::` is the *scope resolution operator*
- `std::cout` means `cout` in the `std` (standard) namespace

Streams

- When you `#include <iostream>` you get three streams to use
 - `std::cout` (short for console output)
 - `std::cin` (short for console input)
 - `std::cerr` (short for console error)

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */

int main() {
    std::cout << "hello, world"; // output
}
```

Insertion
Operator

Output

- To move information from your program to output you use the << (insertion) operator
 - Take info and place it in the output
 - Is a binary operator
 - returns the stream being used
 - Outputs either strings (between " ") or the value in a C++ variable

“ ” vs ‘ ’

- “abc” is a string type (sequence of characters)
- ‘a’ is a character type, a *single* character
- Different than Python!


Semicolons

- C++ uses the ; (semicolon) to terminate a statement
- Not all lines require them
 - Expressions require them
 - Declarations require them
 - Blocks **do not** require them
- We'll get the hang of where they go
- A good editor will help with missing ; and similar issues

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */
Return
type
int main() {
    std::cout << "hello, world"; // output
}
No
semicolon
```



Style Guide

- We select a style for writing our programs so:
 - They are more readable (very important)
 - We have a system amongst ourselves so we can agree on format
 - Be religiously picky about how we do things ;-)

Google

- Why not use Google's Style guide?
- <https://google.github.io/styleguide/cppguide.html>
- They have a very complete style guide
 - When in doubt, consult
- Not a problem for the first project, will be later

Variable names: rules and style

- Variable name rules
 - Only digits, letters, and underscore are allowed
 - Can't start with a digit or underscore
 - Case matters
 - Namespace matters
- Variable name style
 - Lowercase with underscores between words (like in Python)
 - Meaningful / readable
 - Avoid unfamiliar abbreviations

Comments

- Comment at the top of the file
 - Name, date, what it is about
- Variable names shouldn't require comments (descriptive names)
- Functions should have comments
 - Input, output, what it does
- "If it was hard to write, it will be hard to read"
 - Comment the hard parts

First Program: Hello, World

```
#include <iostream>

/*
 * hello world program
 */

int main() {
    std::cout << "hello, world"; // output
}
```