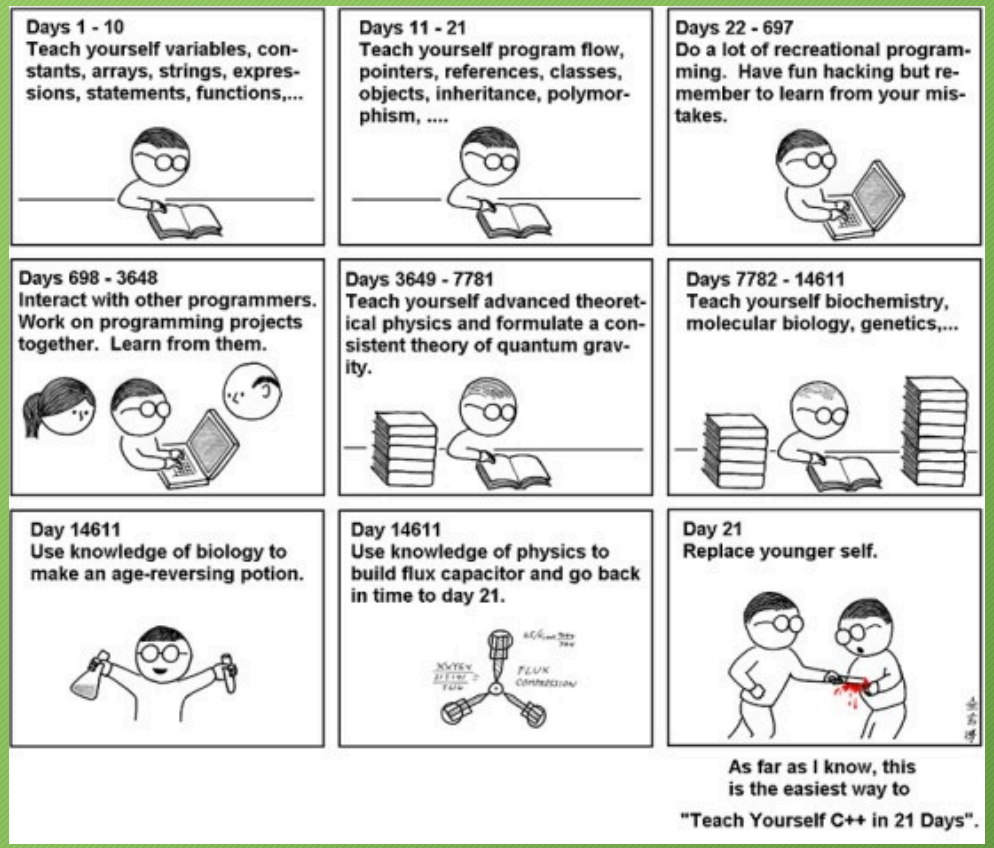
Initial C++ — Session 1

Tristan Brindle

About these sessions

- An introduction to C++
- A mixture of talks, class exercises and homework
- We can’t turn you into an expert (sorry!)
- …but we’ll try to give you enough information to get started

“Teach yourself C++ in 21 days”

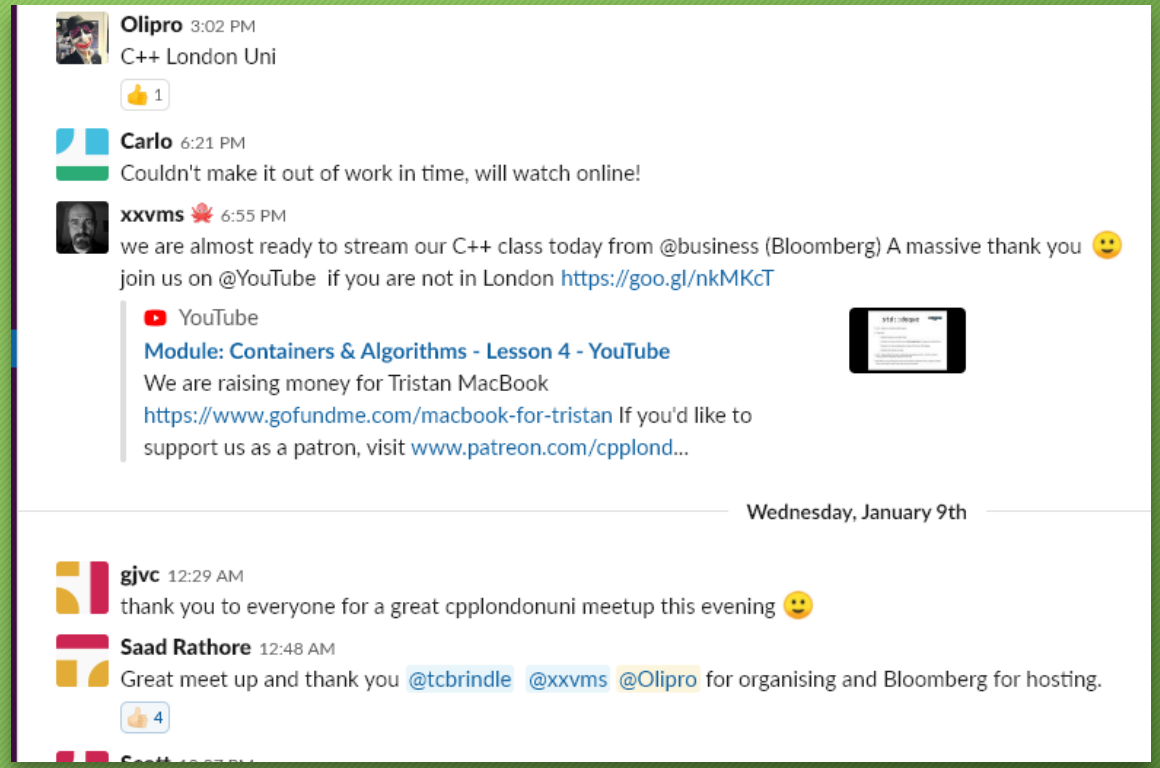


Modules

- Our programme is broken down in to separate modules
- Format: 4-8 weeks of tutorials and practise, followed by a short multiple-choice test to help us gauge everyone's progress
- This module is called "Initial C++"
- No prior C++ knowledge assumed
  - …but prior programming experience will be helpful

Feedback

- We'd love to hear from you!
- The easiest way is via the *CPPLang* Slack organisation. Our chatroom is #cpplondonuni
- If you already use Slack, don't worry, it supports multiple workgroups!
- Go to https://slack.cpp.al to register.



Why C++?

- Usually because it's *fast*
  - Direct access to hardware
  - Zero-overhead abstractions
  - Efficient resource usage

- It's used everywhere
  - Everything from micro-controllers to supercomputers
  - Games, financial trading, web browsers, etc etc…

Why *not* C++?

- Usually because it's *hard*
- Partly true unfortunately
  - C++ allows access to low-level facilities
  - C++ has lots of features — learn use them wisely!
  - Some warts and "gotchas" due to its age

- …but it's not *that* hard! 🙂

A (very) brief history of C++

- 1979: Bjarne Stroustrup starts work on “C with Classes”
- 1983: C with Classes renamed C++
- 1990: ISO committee formed to standardise C++
- 1998: First standard version released (C++98)
- 2011: Major update to the standard (C++11)
- 2014, 2017: Further standard updates (C++14, C++17)
- 2020, 2023….?

“Modern C++”

- C++11 changed the game dramatically
- Don’t bother learning C++98!
- We *strongly* encourage you to do your own reading!
  - …but make sure any textbooks or online resources you use are teaching you *today’s* C++.


C++ London
Uni

Exercise 1

- Go to wandbox.org
- Enter this text (delete what's in the box if necessary)
- Click "run"

```
// Our first C++ program!

#include <iostream>

int main()
{
    std::cout << "Hello world\n";

    return 0;
}
```

Deconstructing “Hello World”

```
// Our first C++ program!
```

- This is a comment
- Inline comments start with two slashes (//) and continue to the end of the line
- Multiline comments start with /* and end with */

Deconstructing “Hello World”

```
#include <iostream>
```

- This line tells the compiler to include the contents of the `iostream` header in our program
- `iostream` is provided by the standard library and contains code to let us write to (and read from) the console
- `#include` is used to break large programs into smaller, manageable pieces, and to use code from other libraries (as we’ve done here)

Deconstructing “Hello World”

```
int main()
```

- This line declares a *function* called “`main`” which returns an `int`(-eger) and takes no parameters
- Every C++ program contains a `main()` function, which is where execution starts.
- The `main()` function is a bit special!

Deconstructing “Hello World”

- A curly brace opens a *block*
- In this case, the block contains the definition of our `main()` function
- Blocks control the *lifetime* of variables in C++, as we’ll see later

Deconstructing “Hello World”

```
std::cout << "Hello world\n";
```

Deconstructing “Hello World”

```
std::cout << "Hello world\n";
```

- `cout` (“console output”) is an object provided by the standard library for printing text
- As part of the standard library, it belongs to the `std` *namespace*, so we write `std::` to access it
- Later we’ll see a shortcut to avoid having to type `std::` everywhere, but use it with caution.

Deconstructing “Hello World”

```
std::cout << "Hello world\n";
```

- The << symbol means (in this case) “pass the thing on the right to the output stream on the left”
- This is an example of *operator overloading* in C++
- Later, we’ll see other meanings of <<, and how to define the meaning of operators for our own types

Deconstructing “Hello World”

```
std::cout << "Hello world\n";
```

- This is an example of a *string literal*
- The `\n` at the end means “start a new line here”
- Sometimes you’ll see `std::endl` used as an alternative way to start a new line

Deconstructing “Hello World”

```
std::cout << "Hello world\n";
```

- Most C++ statements end with a semicolon
- Think of it as being like the full stop at the end of a sentence
- If you forget it, the compiler will usually tell you…
- …but if you get strange errors, check that you’ve got your semicolons correct

Deconstructing “Hello World”

```
return 0;
```

- The `return` keyword tells the program to leave the current function, returning the given value (in this case `0`) to the caller
- By convention, returning zero from `main()` tells the operating system that the program ran successfully
- Any other value indicates an error

Deconstructing “Hello World”

```
}
```

- This closes the block we opened earlier
- When we leave a block, local variables defined in that block get destroyed
- This is the single best thing about C++ (really!)


C++ London
Uni

The C++ compilation process

- C++ is a *compiled language*
- A program called a *compiler* takes our source code and turns it into an *executable* which we can run
- Many potential errors in our program can be caught by the compiler — we call these *compile-time errors*
- The compiler will also generate *warnings* for code which is legal but potentially buggy — don't ignore these!
- Wandbox is an example of an online compiler — it compiles and runs the generated executable in a single step

The C++ compilation process

- Schematically, the process looks like this:


C++ London
Uni

Types

- In programming languages, a *type* is a way of giving meaning to some data
- The type of some data tells us *what it represents* and *what we can do with it*
- For example, we can multiply two numbers, but we cannot meaningfully “multiply" two strings
- The C++ language provides several built-in (“fundamental”) types which we can use in our programs, and many facilities for defining our own types
- The standard library also provides many useful pre-defined types

Types

- C++ is a *statically-typed* programming language
- This means that every piece of data used by a program has its type determined when the program is compiled
- C++'s *type safety* means that we will be can only perform operations which make sense for the types involved
  - This allows many potential errors to be caught by the compiler, before the program even runs
- There are ways around the type safety rules, but avoid them if at all possible

Some useful types

- `int`: represents an integer (whole) number, e.g 0, 1, 12345, -54321
- `float`: represents a real number, e.g 3.142, 2.718, -123.456
- `bool`: represents a boolean value, `true` or `false`
- `char`: represents a single (ASCII) character, e.g. a, B or ?
- `std::string`: represents a sequence of characters, e.g. “Hello”
- `std::vector`: represents a sequence of values of some type, e.g. [1, 2, 3, 4] or [0.0, -0.1, -0.2, -0.3]
  - This is an example of a *generic type*

Functions

- As in most languages, in C++ programs are built up out of *functions*, small pieces of reusable code
- We've already seen `main()`, a special function where execution of our program always begins
- [Note: unlike some other languages, C++ has both "methods" (functions which belong to a class) and "free-standing" functions; today we are talking about the latter.]

Functions

- In the most general case, a function takes some *input data*, performs some action(s), and returns some *output data*
  - However, it's also common to have functions which do not take any input, or functions which do not return any output
- In C++, a function may take *any number* of input values
  - We call these the *function parameters*
- A function may return *zero or one* output values
- We need to tell the compiler the *data types* of a function's input and output

Functions

- Here is an example of a function named `add`
- We can see it takes two parameters, both of type `int`, which we have named `a` and b
- This function *returns a value* of type `int` as well
- In the *body* of the function, we perform the actual calculation

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

Functions

- Here is another example of a function
- This function takes a single parameter of type float
- This time, the function does not return any data, so we write its *return type* as `void`
- `void` is a special type meaning “does not return anything”

```
void print_float(float f)
{
    std::cout << f;
}
```

Calling functions

- We can *call* (execute) a function from another part of our source code by writing its name followed by its inputs (if any) in brackets
- For example `print_float(3.142f)` or `add(4, 5)`
- Note that in C++, functions must be *declared* before they may be used
- This means that you need to write the definition of your functions first, before the point at which the function is used!

Example

```cpp
#include <iostream>

void print_int(int i) { std::cout << i << '\n'; }

int get_one() { return 1; }

int add(int a, int b) { return a + b; }

int main()
{
    print_int(3);

    print_int(add(4, 5));

    print_int(add(get_one(), get_one()));

    return 0;
}
```

Exercise 2

- In your “hello world” program in Wandbox, write a function

```
void hello_cpp_london_uni()
```

- which prints “Hello C++ London Uni” to the console
- Call this function from your `main()` function

Solution

```
void hello_cpp_london_uni()
{
    std::cout << "Hello C++ London Uni\n";
}

int main()
{
    hello_cpp_london_uni();

    return 0;
}
```

Exercise 3

- In Wandbox, write a function `say_hello()` which takes a parameter of type `std::string` called name, and returns a string containing that name with "`Hello `" in front
- Use this function to print "Hello <your name>" from `main()`, e.g. "Hello Tristan"
- You will need to add `#include <string>` near the top of your program to use `std::string`

Solution

```
#include <iostream>
#include <string>

std::string say_hello(std::string name)
{
    return "Hello " + name;
}

int main()
{
    std::cout << say_hello("Tristan") << '\n';
    return 0;
}
```

Summary

- This was only a very brief introduction to the wonderful world of C++
- We've learnt how "hello world" works
- We've learnt about types and type-safety
- We'v learnt how to `#include` standard library headers
- We've learnt how to define our own functions
- We've been introduced to std::string

Homework

- Read about `std::cin`, the input counterpart to `std::cout`
- Write a program which reads in a string using `std::cin`, and then prints “Hello, <name>”, where <name> is the string the user supplied
- Bonus: use `std::vector` to read in a list of strings. When an empty string is entered, print “Hello <name>” for each string that was entered so far, and then exit the program.

Thank You!

As usual, we will be going to the pub! Support us @ https://patreon.com/CPPLondonUni