Initial C++ — Session 2

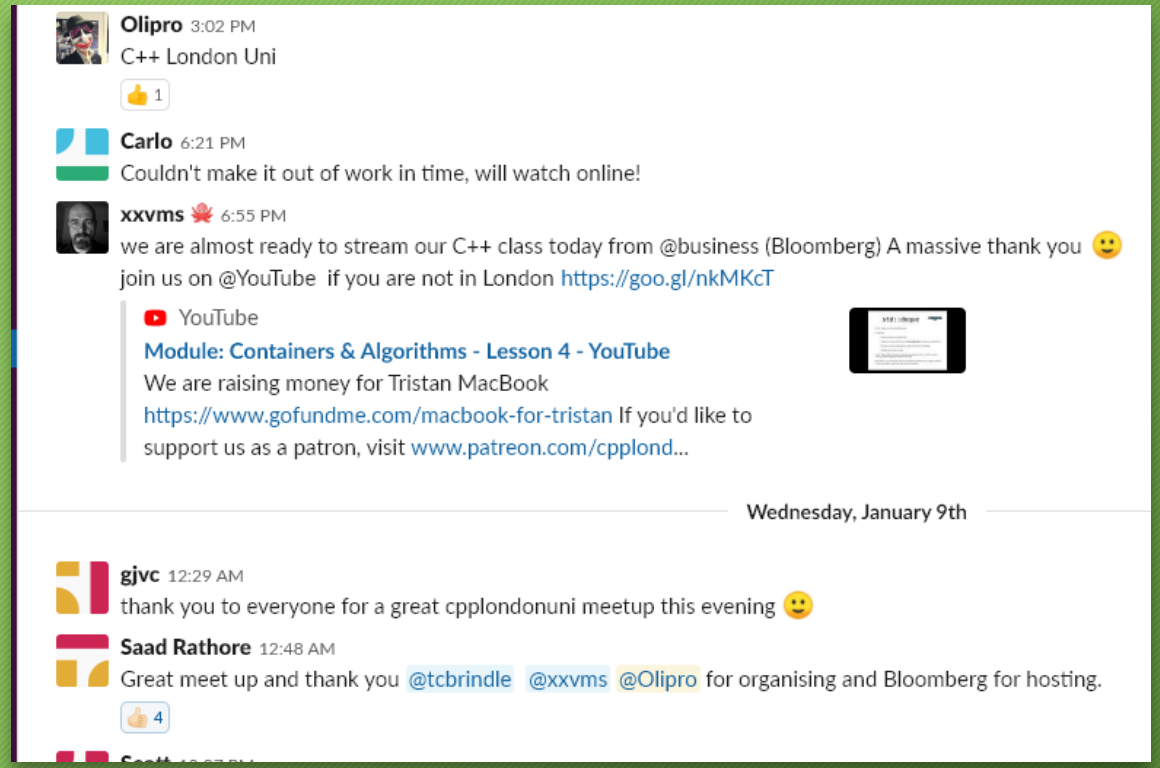
Tristan Brindle

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

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.



Last week

- First introduction to C++
- Types in C++
- Defining and calling our own functions

This week

- Revision from last week
- Introduction to using variables
- Passing variables to functions
- If statements

Revision: types

- A *type* is a way of assigning meaning to some data
- The type of some data tells us *what it represents* and *what we can do with it*
  - Just like e.g. PDF files or MP3s on your computer
- C++ is a *statically-typed* programming language
- Every piece of data we use in our program has a type
  - That type **does not change** for the life of the program
- C++ has *type-safety* rules to help prevent mistakes

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*

Revision: functions

- A *function* is a list of instructions which we can *call* (execute) from another part of our source code
  - Some languages call them subroutines, procedures or methods
- Functions can take zero or more pieces of input data
  - These are called the function's *parameters*
  - We provide these pieces of data when we call the function
- Functions can optionally *return a value* to the caller

Revision: functions

- When we write a function, we need to specify the *type* of *each parameter*
  - Each parameter may also have a *name* which we can use to refer to it
- We also need to specify the type of the *return value*
  - If a function does not return a value, we use the special return type `void`
- For historical reasons, in C++ function declarations need to appear *earlier in the source code* than where they are called

Revision: functions

- To define a function, we write its *return type*, followed by its *name*, followed by its parameters in `(  )` brackets
- After that, we write the *body* of the function (the list of instructions it needs to perform) in `{ }` braces
- If the function returns a value, we **must** use a *return statement* to exit the function (e.g. `return 4;`)
- It's always a mistake to "fall off the bottom" of a function which does not return `void`!
  - (But special rules exist for the `main()` function)

Revision: functions

```
int add(int i, int j) { return i + j; }
/* Name: add
 * Return type: int
 * Parameters: int, int
 */

void print(float f) { std::cout << f << '\n'; }
/* Name: print
 * Return type: <does not return a value>
 * Parameters: float
 */

int main() {
    print(3.14f);
    return add(3, 4);
}
```

Last session's exercise

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

```
void hello_cpp_london_uni()
```

- which prints "Hello C++ London Uni"
- Call this function from your `main()` function

My solution

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

int main()
{
    hello_cpp_london_uni();

    return 0;
}
```

Variables

Variables

- Our programs would not be very interesting if they could only consist of functions!
- To be useful, we also need to use *variables*
- A variable is, very roughly, an *item of data* that has a *name*
- As with all data in C++, every variable we use has a *type*
- The data currently held in a variable is called its *value*
- The value of a variable may change as our program runs

Variables

- To declare a variable, we write its *type*, followed by its *name*, followed by an *initialiser*
- Here we are creating a new int variable named `my_int` with value 3
- We can *change the value* of a variable by using assignment (`=`)

```
int main()
{
    int my_int = 3;
    // my_int is an int with value 3

    my_int = 4;
    // my_int now has value 4

    std::cout << my_int << '\n';
    // prints 4

    return 0;
}
```

Variables

- We can use the *result of a function* to initialise a variable
- C++ will check the variable's type and the function's return type are "compatible"
- We cannot use a `void` function to initialise a variable

```
int add(int i, int j) { return i + j; }

std::string get_name() { return "Tom"; }

void print_float(float f) {
    std::cout << f << '\n';
}

int main() {
    int a = add(3, 4);
    // okay, a has value 7

    int b = get_name();
    // ERROR: std::string and int are not compatible

    int c = print_float(3.0f);
    // ERROR: print_float does not return a value

    return 0;
}
```

Copying values

- We can use the name of a variable as the *initialiser* for a new variable
- This creates a *copy* of the original variable
- **Important**: copies are *distinct*!
- Changing one will not affect the other

```
int main() {
    int a = 44;
    // a has value 44

    int b = a;
    // b has value 44

    a = 99;
    // a has value 99

    std::cout << b << '\n';
    // prints 44;

    return 0;
}
```

Variables and functions

- We can use our variables as the input to functions
- **Important**: the function receives a *copy* of each value passed to it!
- If we change the value of a function parameter, that *does **not** affect* the original variable
- This is called *pass-by-value*

```
int add_one_v1(int i) { return i + 1; }

void add_one_v2(int i) { i = i + 1; }

int main() {
    int a = 3;

    int b = add_one_v1(a);
    // b has value 4

    add_one_v2(b);
    // b still has value 4!

    return 0;
}
```

Exercise

- Using Wandbox (wandbox.org):
  - In your `main()` function, create a variable named `first_name` of type `std::string`, and initialise it with your name
  - Print the value of this variable using std::cout
  - (Don't forget to `#include <string>` and `<iostream>`!)
  - Write a function `str_join()` which takes two strings and returns a new string which consists of the two inputs separated by a space
    - Hint: you can use the `+` operator for string concatenation
  - In main(), create a new variable named `full_name` which is the result of calling str_join() with `first_name` and your surname

My solution

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

std::string str_join(std::string a, std::string b) {
    return a + " " + b;
}

int main() {
    std::string first_name = "Tom";
    std::cout << first_name << '\n';

    std::string full_name = str_join(first_name, “Breza");
    std::cout << full_name << '\n';

    return 0;
}
```

If statements

If statements

- One of the basic building blocks of programs is the *if statement*
- This tests some condition, and performs some some instructions *if* the condition is true
- The basic form of an if statement in C++ is

```
if (condition) {
    // do something
}
```

If statements

- For example, we can test whether two variables hold the same value using the == operation
  - Note *two* equals signs!
- There is a corresponding != (not equal) operation which tells you if two variables hold *different* values
- Many types also have <, >, <= and >= operations with their usual mathematical meanings

```
int main() {
    int a = 3;
    int b = 3;

    if (a == b) {
        std::cout << "a equals b\n";
    }

    if (a > 0) {
        std::cout << "a is positive\n";
    }

    return 0;
}
```

If statements

- When writing an `if` statement can also add `else if` to test a second condition:

```
if (condition) {
    // do something
} else if (other condition) {
    // do something else
}
```

- We can have as many `else if` statements as we like
- Conditions are tested *in the order that they appear*

If statements

- Finally, we can add an else statement as a fallback if none of the other conditions are true:

```
if (condition) {
    // do something
} else if (other condition) {
    // do something else
} else {
    // do a third thing
}
```

If statement example

```
void greet(std::string name) {
    if (name == “Tom") {
        std::cout << "Hello Tom";
    } else if (name == "Oli") {
        std::cout << "Hello Oli";
    } else {
        std::cout << "I don't know you";
    }
}

int main() {
    greet(“Tom");

    greet("Steven");

    return 0;
}
```

Exercise: FizzBuzz

- The modulus operation % returns the *remainder* of dividing one integer by another
- This can be used to test whether one integer is *exactly divisible* by another
- An integer is exactly divisible by another if the remainder is zero
- For example, 9 is exactly divisible by 3, and 125 is exactly divisible by 5

```
int i = 16;

if (i % 2 == 0) {
    std::cout << "i is even\n";
} else {
    std::cout << "i is odd\n";
}
```

Exercise: FizzBuzz

- In Wandbox, write a function named `fizzbuzz` which takes an `int` as input
  - If the int is exactly divisible by 3, print “fizz”
  - If the int is exactly divisible by 5, print “buzz”
  - If the int is exactly divisible by *both* 3 and 5, print “fizzbuzz”
  - Otherwise, print “Not fizzy or buzzy”
- Try calling this function from `main()` with different values, e.g. 99, 125, 225, 1024…

My solution

```
void fizzbuzz(int i) {
    if (i % 15 == 0) {
        std::cout << "fizzbuzz\n";
    } else if (i % 3 == 0) {
        std::cout << "fizz\n";
    } else if (i % 5 == 0) {
        std::cout << "buzz\n";
    } else {
        std::cout << "not fizzy or buzzy\n";
    }
}

int main() {
    fizzbuzz(99);  fizzbuzz(125);
    fizzbuzz(225); fizzbuzz(1024);

    return 0;
}
```

Thank You!

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