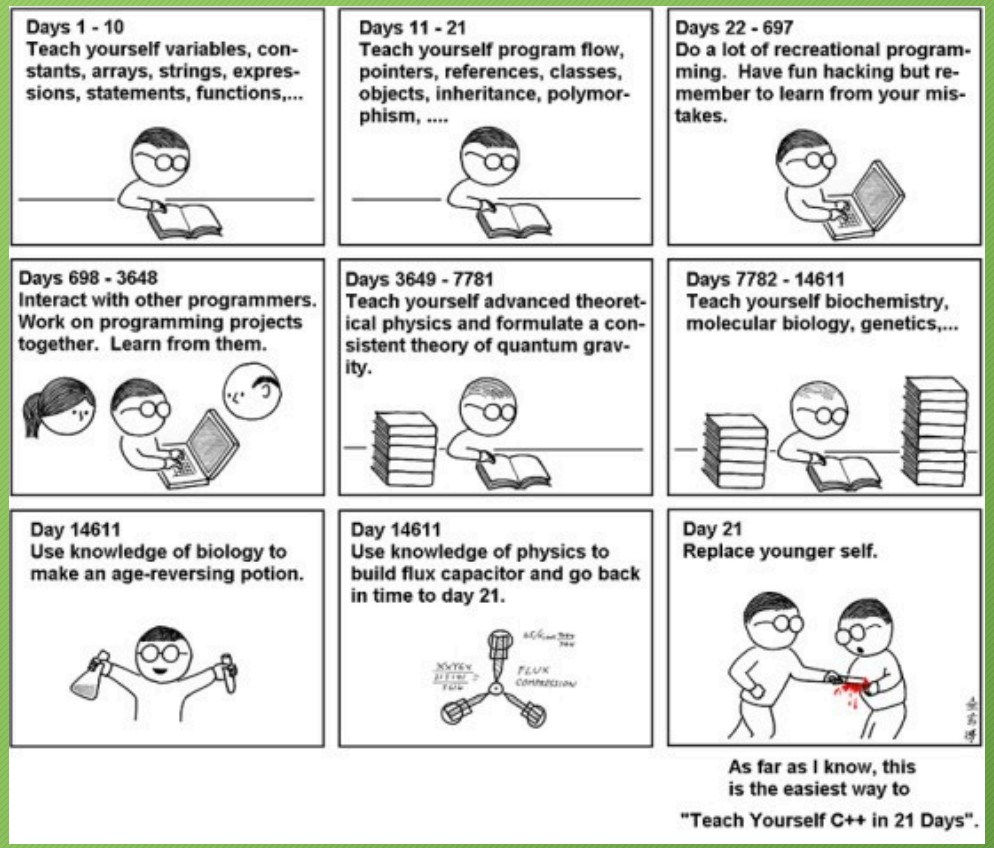
Initial C++ — Session 1

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 sessions 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

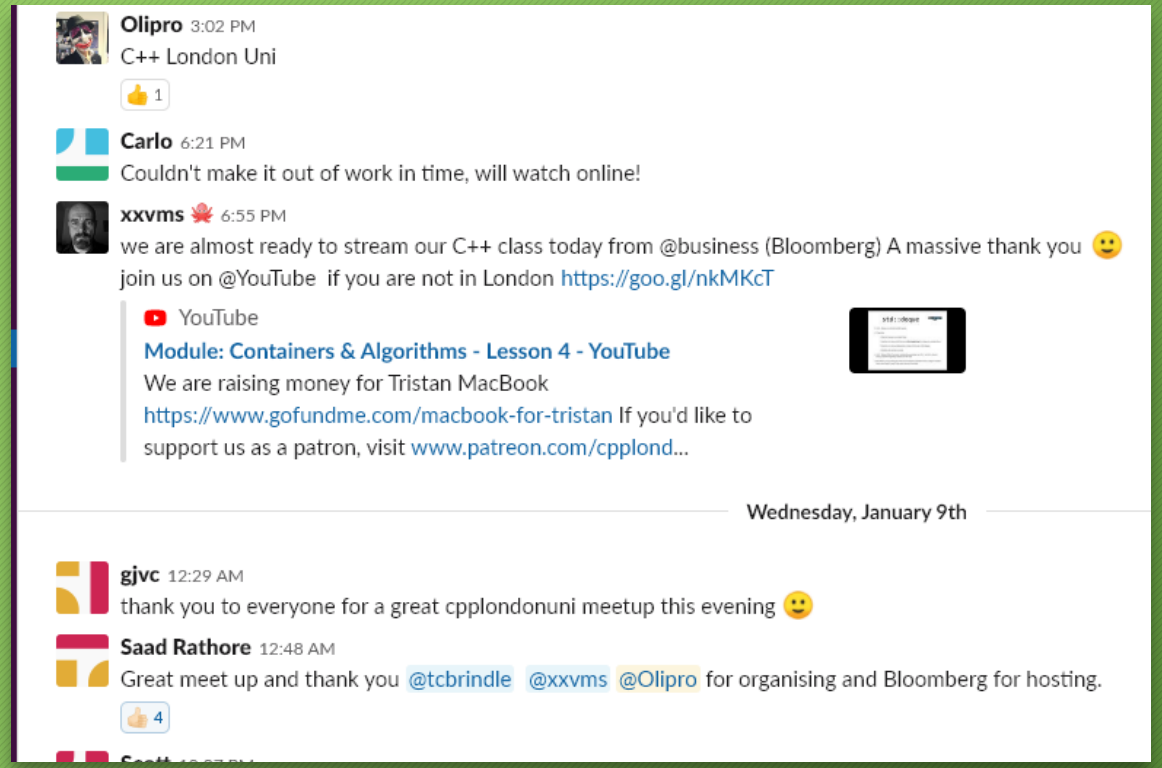
Course materials

- All of our sessions are live-streamed on our YouTube channel, youtube.com/cpplondonuni
- The recordings are available to watch later
- If you can't make it to a session, this is the best way to catch up :)

- These slides will be available after the session on our Github page, github.com/cpplondonuni

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

Your first C++ program

- 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;
}
```

Your first C++ program

- This program just prints “Hello world” to the screen, followed by a new line
- It’s the traditional “first program” in any programming language
- You’re not expected to know what it all means at this stage!
- But it will give you a taste of what C++ code looks like
- Things will become clearer as the course progresses!

The C++ compilation process

- C++ is a *compiled language*
- A program called a *compiler* takes our source code (a text file) and turns it into a program which your computer can run
- Many potential errors in our program can be caught by the compiler — we call these *compile-time errors*
- 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:

Comments

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

- This is a comment
- Comments are for information only, and are ignored by the compiler
- Comments start with two slashes (//) and continue to the end of the line
- You can also write multi-line comments in a block of text that starts with /* and ends with */

Semicolons

```
return 0;
```

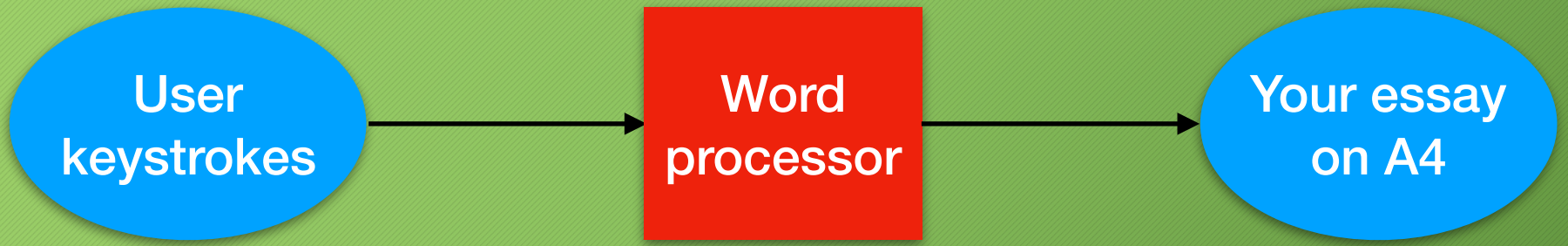
- Most statements in C++ end with a semicolon ;
- This is like the full stop at the end of a sentence
- Forgetting the semicolon is a very easy mistake to make!


C++ London
Uni

Question

- Q: What is the *purpose* of a computer program?
- A: To take some *input* and transform it into useful *output*

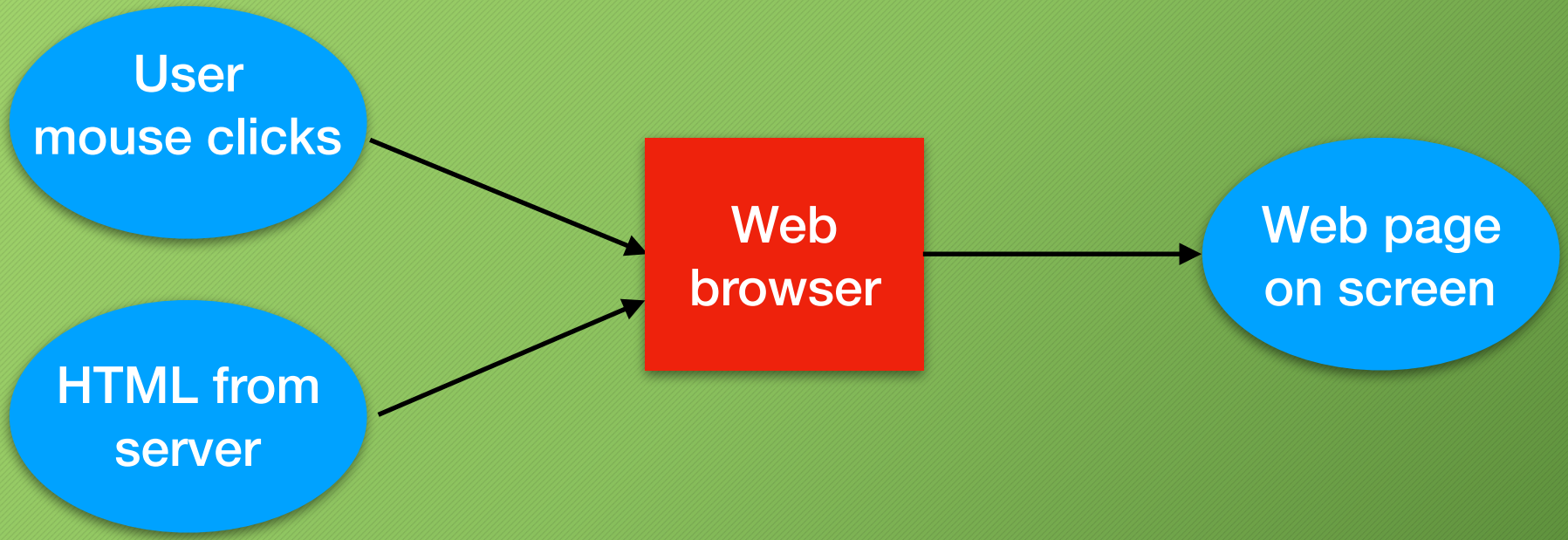
Example programs



Example programs

Example programs

Example programs



Example programs

- A program is “just” a *list of instructions* for the computer to execute
- We write these instructions in C++, and the compiler turns them into *machine code* instructions that the CPU understands
- A large program like a web browser will have billions of these instructions
- How do we manage this complexity?

Subprograms

- We can manage the complexity by breaking our program down into smaller *subprograms* or *subroutines*
- A subprogram takes some *input*, executes a *list of instructions*, and produces some *output*
- These subprograms can again be broken down into smaller sub-programs, and so on
- Eventually we end up with a large number of small subprograms each of which are (hopefully) easy to write and easy to understand

Subprograms => functions

- In C++, subprograms are called *functions*
- A C++ function takes some *input*, executes a *list of instructions*, and produces some *output*
- Every function has a *name*, which we use to refer to it in our source code
- When we want to *run* a subprogram, we supply it with input, wait for its instructions to be executed, and then get back its output
- In C++ this is known as *calling* a function

The `main` function

- The most important function in any C++ program is called `main`
- This is “main program”, rather than a subprogram
- When we run our program it starts executing the list of instructions in the `main` function, and ends when `main` has finished
- If it helps, you can think of `main` as a function that gets “called” by the computer’s operating system to start the program
- `main` always returns an integer status code back to the OS
    - By convention, zero means “everything was fine"

The main function

- Let's look back at our "hello world" example

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

#include <iostream>

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

    return 0;
}
```

The `main` function

- Let's look back at our "hello world" example
- This section is the *definition* of the `main` function
- The *list of instructions* to execute appears between the { and }
- At the end, we tell the function to return the value `0` back to the operating system

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

#include <iostream>

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

    return 0;
}
```


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 common types which we can use in our programs
- There are also many facilities for defining our own types later

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*

Writing functions

- Here is an example of a function named `add`
- It takes two items of input, 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 a value”

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

int main() {
    print_int(3);

    print_int(add(4, 5));

    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”
- Call this function from `main` function

Solution

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

int main()
{
    hello_cpp_london_uni();

    return 0;
}
```

Homework

- 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`

Thank You!

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