C++ London University Session 2

Tristan Brindle

Gentle reminder

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- If you don’t know, please **ASK**

Lesson Plan

- Getting set up with CLion
- Any questions from last week's material?
- "Homework" discussion
- Declaring variables, `const` and `auto`
- `If` statements
- Fizzbuzz
- Function declarations and definitions
- Header files and implementation files
- References and `const` references

Getting set up with CLion

- https://www.jetbrains.com/clion/

- If you need it https://nuwen.net/mingw.html

Getting set up with CLion

- Live demo — creating a new project

CMake

- CLion uses the CMake *build system*

- A build system takes care of calling the compiler with the right flags when you change source files

- CMake is the most common build system for C++ projects, and is widely supported by IDEs (CLion, QtCreator, Eclipse, Visual Studio 2017, …)

- There are many, many others too (MSBuild, Autotools, QMake, Waf, Scons, etc, etc…)

Is everyone ready to go?

Any questions from last week's material?

“Homework” problems from last week

“Homework” problems from last week

- Read about `std::vector`. Modify your solution to Exercise 3 to print “Hello Tom”, “Hello Phil”, “Hello Tristan” on separate lines using a vector of strings and a range-for loop.

“Homework” problems from last week

- My solution:

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

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

int main()
{
    const std::vector<std::string> names = {
        "Tom", "Phil", “Tristan"
    };

    for (const auto name : names) {
        std::cout << say_hello(name) << '\n';
    }
}
```

“Homework” problems from last week

- Write a program to ask the user to enter their name at the console. Read this into a `std::string` using `std::cin`. If the name is one of “Tom”, “Phil”, “Tristan” or your name then print “Hello <name>!” (e.g. “Hello Tom!”), otherwise print “Hello stranger!”

“Homework” problems from last week

- My solution:

```
#include <iostream>

int main()
{
    std::cout << "Please enter your name:\n";
    std::string name;
    std::cin >> name;

    if (name == "Tom"   ||
        name == "Phil"  ||
        name == "Tristan") {
        std::cout << "Hello " << name << "!\n";
    } else {
        std::cout << "Hello stranger!\n";
    }
}
```

Any questions before we move on?

Variables

- Dictionary definition: (roughly) “a named storage location for some data”
- In C++, every variable has a *type*, which dictates what sort of data it can hold
- The data currently held in a variable is called its *value*
- In C++, the *lifetime* of a variable is usually tied to the scope (block) in which it is declared

Declaring Variables

- To declare a variable, we can say

```
type-name variable-name = initialiser;
```

- e.g.

```
int i = 0;
```

- (There are a couple of other initialisation forms we'll see later when we discuss classes)

- **Always** initialise your variables

Declaring Variables (2)

- C++11 added *type deduction*, so we could also say

```
auto variable-name = initialiser;
```

- e.g.

```
auto i = 0;
```

- Now the type of `i` is determined by its initialiser (still `int` in this case).

- This can be really handy, but (as ever) use with caution

Constants

- We can declare a variable to be a constant using the keyword `const` in front of the type name, for example

```
const int i = 0;
```

- When declared like this, the value of `i` cannot be changed after it is initialised

- This is helps reduce programming errors and (sometimes) allows better optimisation

- Pro tip: make variables “const by default”, mutable only when necessary

Variables

- Live demo: variable declarations, `const` and `auto`

Variables

- Exercise: experiment with variable declarations
    - What happens if you try to modify a const variable?
    - What happens if you declare two variables with the same name?
    - What happens if you declare a variable in a block with the same name as a variable in an outer block?

Any questions before we move on?

`if` statements

- One of the basic building blocks of programs is the `if` statement

- The basic form of an if statement is

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

`if` statements

- We 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
}
```

Exercise: fizzbuzz

- The modulus operator % returns the remainder after dividing one integer by another

- This can be used to test whether one integer is divisible by another

- For example

```
const int i = 16;

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

Exercise: fizzbuzz

- Exercise:
  - Create a new project “fizzbuzz” in CLion
  - Use std::cin to ask the user to input a number
  - If that number is divisible by 3, print “fizz”.
  - If the number is divisible by 5, print “buzz”.
  - If the number is divisible by both 3 and 5, print “fizzbuzz”.
  - If the number is not divisible by either 3 or 5, print “not fizzy or buzzy”

Exercise: fizzbuzz

- Solution:

```
#include <iostream>

int main()
{
    std::cout << "Please enter a number:\n";

    int i = 0;
    std::cin >> 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";
    }
}
```

Any questions before we move on?

Definitions and declarations

- Before we can call a function in C++, the compiler must have seen a *declaration* of that function.

- A function declaration describes the “signature” of the function — the parameter types it takes, and its return type.

- For example

```
std::string say_hello(std::string name);
```

Definitions and declarations

- A function *definition* is a *declaration* followed by the function *body*

- A function can be declared many times, but there can only ever be one *definition*

Definitions and declarations

- Exercise:
    - Create a new project in CLion. Write a function “*say_hello()*” which just prints “Hello World”;
    - Add a declaration of the *say_hello()* function
    - Experiment with moving the location of the *say_hello()* definition. What happens if you place it after *main()*? What happens if you then comment out the declaration? Why?
    - What happens if you comment out the definition of *say_hello()*, but leave the declaration? Why?
    - What happens if you have two definitions of the *say_hello()* function? Why?

Definitions and declarations

- Solution: live demo

Any questions before we move on?

Header files and implementations files

- It's normal to split large projects up into separate files of manageable size

- Problem: how do we call functions defined in other source files?

- Solution: header files

- Headers (usually with `.hpp` extension) contain the declarations of public functions (and types) in the corresponding implementation (`.cpp`) file

Header files and implementation files

- We can include a header file in an implementation file using the `#include` command

- For headers in our own project we need to say

```
#include "header.hpp"
```

- For headers from other libraries we need to say

```
#include <header.hpp>
```

Header files and implementation files

- Header files can themselves `#include` other headers that they rely on

- To prevent errors, headers normally have *include guards* prevent their contents appearing more than once in an implementation file

- Don’t worry about this too much just now: CLion will add include guards for you, or you can say `#pragma once` at the top of your .hpp file.

Header files and implementation files

- Demo: creating a header and implementation file in CLion

Header files and implementation files

- Exercise:
  - Create files `say_hello.hpp` and `say_hello.cpp` in your CLion project
  - Move the definition of *say_hello()* into say_hello.cpp. Add a declaration of this function to `say_hello.hpp`.
  - Modify the code so that *main()* can still call *say_hello()*

Header files and implementation files

- Solution: live demo

Any questions before we move on?

References

- C++ uses *value semantics* by default. This means (roughly) that copies of variables are distinct; changing the value of a copy will not affect the original variable
- We can also declare variables which are *references*.
- A reference is a way of referring to the original variable by a new name
- A reference must be initialised with a variable name. Once created, a reference can never change which variable it refers to.

References

- We declare a reference by saying

```
type& name = variable;
```

- Changing the value of a reference will change the value of the original variable

- Functions can also take arguments *by reference*

- *Be careful when returning references from functions*

References

- Demo: declaring references, *increment()*

References

- Exercise:
  - Experiment with using references
    - What happens if you don’t initialise a reference?
    - Implement the *increment()* function you’ve just seen. What happens if you remove the & from the function parameter? Why?

Const references

- We can also declare a reference as `const`

```
const type& name = variable;
```

- A const reference means that the value of a variable cannot be changed *using that reference*

- Const variables can only bind to const references

- Non-const references are sometimes called *mutable references*

Const references

- References allow us to avoid copying variables
- Use const references whenever you can, mutable references only when you need to change a value
- Most of the time, your function parameters should be const references

Const references

- Exercise
  - Experiment with const references
    - What happens if you try to modify a const reference?
    - What happens if you try to bind a non-const variable to a const reference? What happens if you then modify the value of the original variable?
    - What happens if you try to bind a const variable to a non-const reference?

Const references

- Solution: live demo

Any questions before we wrap up?

Summary

- Getting set up with CLion
- Any questions from last week's material?
- "Homework" discussion
- Declaring variables, `const` and `auto`
- `If` statements
- Function declarations and definitions
- Header files and implementation files
- References and `const` references

Next time

- All about types
  - Defining our own types
    - Member functions
    - Access specifiers
    - Special member functions
    - (Maybe) Inheritance

“Homework”

- What is wrong with the following function definition? How can we fix it?

```
std::string& get_hello()
{
    std::string str = "Hello";
    return str;
}
```

- Write a function `fib(int n)` returning a vector of integers containing the first n Fibonacci numbers

- Extension: modify `fib()` to allow the user to pass the initial “seed” values, defaulting to 0 and 1

- Extension(2): In CLion, create a `libfibonacci` library containing your fib function, and a test program which ensures that the results are correct

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Github: github.com/CPPLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #cpplondon
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂