Session 4

Tristan Brindle

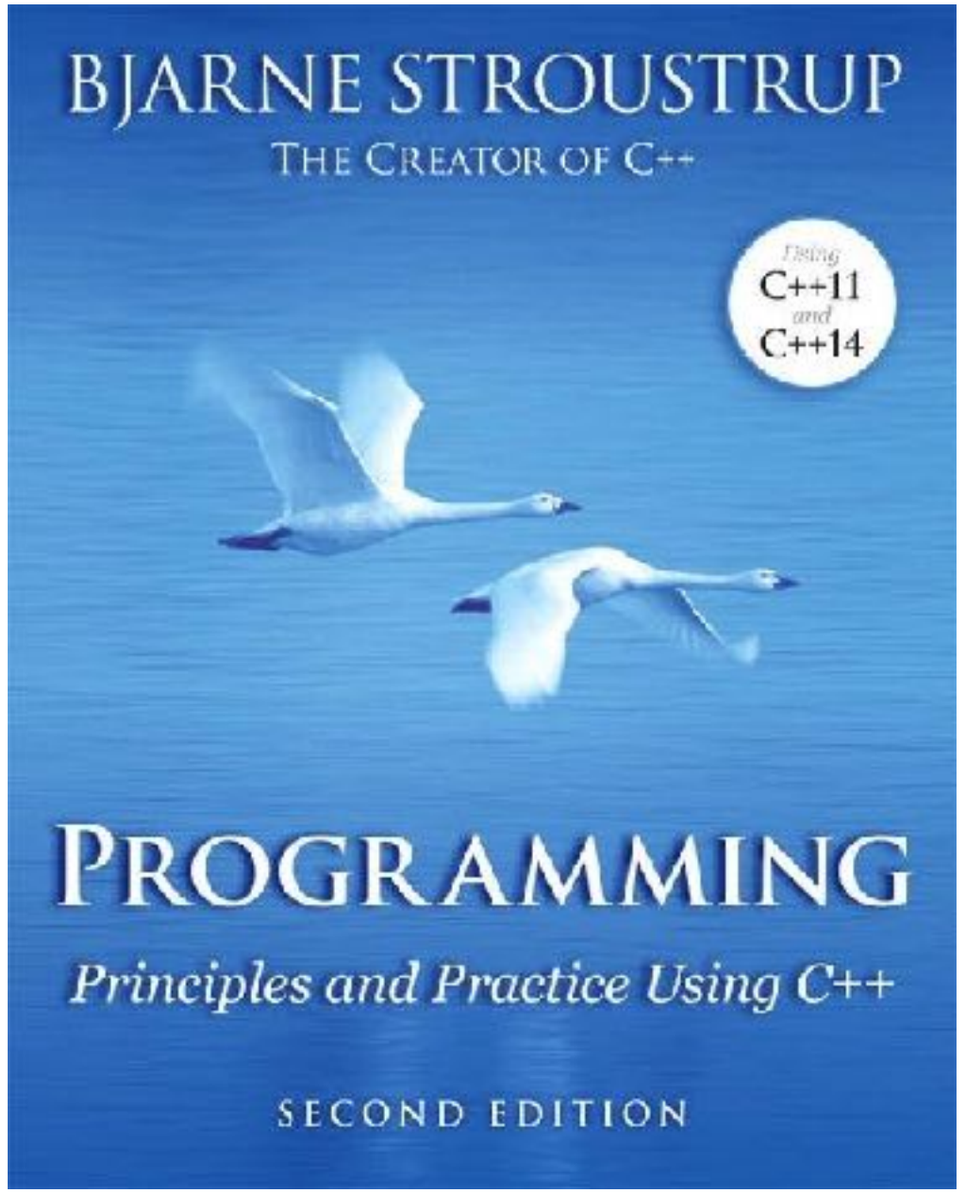
Feedback

- We love to hear from you!
- The easiest way is via the *cpplang* channel on Slack — we have our own chatroom, *#ug_uk_cpplondonuni*
- Go to https://cpplang.now.sh/ for an “invitation”

Textbook

- We'll be using *"Programming Principles and Practice Using C++"* by Bjarne Stroustrup

- Please pick up a copy if you haven't already



Last week…

- Getting set up with a compiler and CLion
- The C++ compilation model
    - What compilers do
    - Using the compiler from the command line
    - The C++ compilation process
    - Separate compilation
    - Using libraries
    - Header files and `#include`

This week

- Value semantics
- Pass by value
- Scope
- Basic object lifetimes
- Basic control flow
  - If statements
  - Loops

Value semantics

- Unlike many other programming languages, C++ uses *value semantics* rather than *reference semantics* by default

- This means (roughly) that copies of variables are distinct; changing the value of a copy will not affect the original variable (i.e. copies are “deep”).

- We’ll see next week how we can use references in C++

Value semantics example


```cpp
int a = 1;

int b = a;
// b has value 1

a = 42;
// a has value 42
// b still has value 1

std::string s = "Hello";
std::string s2 = "World";

s += s2;
// s now has value "HelloWorld"
// s2 still has value "World"
```

Passing by value

- By default, C++ passes arguments to functions *by value*
- This means that the function receives an independent *copy* of the object passed to it
- Modifying a function parameter taken by value will not change the original object
- Similarly, C++ (conceptually) *returns a copy* of an object from a function by default

Passing by value example


```
/* declaration in a header file */
int add_one(int i);


/* ..somewhere else... */

int j = 42;
// call our add_one function
auto ret = add_one(j);
// add_one operates on a *copy* of j
// our j still has value 42
// ret has value 43



/* definition of add_one, in an implementation file */
int add_one(int i)
{
    i += 1;
    return i;
}
```

Exercise

- Create a new project in CLion
- Write a function named `say_hello` which takes a parameter of type `std::string` called *name,* and returns a `std::string`
- Your function `say_hello` should modify *name* by adding the string “Hello “ to the front of it, and then return *name*
- In main, create a `std::string` variable named *me*, initialised with your name. Call your `say_hello` function with *me* as an argument
- Use `std::cout` to show that calling the function has not modified the original variable

My solution


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

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


int main()
{
    std::string me = "Tristan";
    std::string hello_me = say_hello(me);

    std::cout << hello_me << '\n';
    std::cout << me << '\n';
}
```

Any questions before we move on?

Variable scope

- A *scope* is a region of source text in our program
- Every variable has a *scope* in which it lives
- The scope of a variable generally lasts from the point at which we declare it until we leave the *block* (pair of curly braces) it was declared in
- We cannot access a variable outside of its *scope*

Variable scope example


```
void func(int i) // i is in scope
{
    i = 0;
    int j = 1; // j is in scope

    { // open new scope
        int k = 42; // k is in scope

        i = k;      // i is still in scope
        j = k + 1;  // j is still in scope

    } // k's scope ends

    k = 0; // ERROR - k is no longer in scope

} // i, j scope ends
```

Variable lifetimes

- Unlike many other languages, C++ does not use *garbage collection* to manage memory

- Instead, resources like memory are usually managed using *scopes*

- This is a central concept in modern C++, and goes by the ~~silly~~ acronym *RAII*

Variable lifetimes

- When we declare a variable inside a function, it is called an *automatic variable*
- An automatic variable is *destroyed* when we leave the *scope* in which it is declared
- Function parameters are also automatic variables
- Later, we'll see how to declare special functions called *destructors* which can be used to "clean up" when a variable is destroyed

Lifetimes example


```
void function(int i)
{
    int j = i; // j's lifetime begins

    {
        int k = j + 1; // k's lifetime begins
        if (k == 1) {
            // if we leave the function here,
            // i, j and k are destroyed
            return;
        }
    } // k is destroyed here

} // i and j are destroyed here
```

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

- In CLion, create a new project called `fizzbuzz`
- Create a new source file `fizzbuzz.cpp` and an accompanying header file
- In `fizzbuzz.cpp`, write a function `void fizzbuzz(int i)` which performs the following using `if` statements:
    - If `i` is divisible by 3, print “fizz”.
    - If `i` is divisible by 5, print “buzz”.
    - If `i` is divisible by both 3 and 5, print “fizzbuzz”.
    - If `i` is not divisible by either 3 or 5, print “not fizzy or buzzy”
- Add a declaration of your `fizzbuzz` function to the `fizzbuzz` header
- In your main() function, test calling fizzbuzz() with a variety of inputs

Fizzbuzz: solution


```hpp
// fizzbuzz.hpp

void fizzbuzz(int i);
```

```cpp
// fizzbuzz.cpp

#include "fizzbuzz.hpp"
#include <iostream>

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";
    }
}
```

```cpp
// main.cpp

#include "fizzbuzz.hpp"

int main()
{
    fizzbuzz(9999);
    fizzbuzz(125);
    fizzbuzz(225);
    fizzbuzz(1024);
}
```

Loops

- Another of the fundamental building blocks of programs are *loops*

- A *loop* tells the program to keep repeating some action until a *condition* fails

- C++'s loop keywords are very similar to those found in other languages

While loops

- The simplest form of loop uses the `while` keyword
- We write this as

```
while (some_condition) {
    // do_something
}
```

- The program keeps repeating the *body* of the loop until the *condition* is `false`
- If the condition is `false` to start with, we will never enter the loop

Do-while loops

- Similar to a while loop is a do-while loop. This is written

```
do {
    // something
} while (condition);
```

- Unlike a plain while loop, the body of do is executed once *before the condition is checked*

- This means that a do-while loop will execute at least once

For loops

- The most common type of loop is the *for loop*
- This has the form

```
for (init ; condition ; increment) {
    // do something
}
```

- First the *init* part is run, then the *condition* is checked
- If the *condition* is `true`, the body of the loop is executed, followed by the *increment* step
- This is repeated until the *condition* is `false`

For loop example


```
int main()
{
    for (int i = 0; i < 10; i++) {
        std::cout << "i is " << i << '\n';
    }
}
```

Range-for loops

- C++ also has range-for loops, which can be used to loop over collections (*ranges*)

- This has the form

```
for (declaration : range) {
    // do something
}
```

- This is extremely useful when dealing with vectors and strings!

- A range-for loop can be used with any type that meets the *range concept* requirements, including all standard library collections

Range-for example


```
int main()
{
    std::vector<int> vec{1, 2, 3, 4, 5};

    for (int i : vec) {
        std::cout << i << '\n';
    }
}
```

Exercise

- Complete exercise 8 of chapter 4 of the textbook

- This is a bit tricky! Try working out the problem with a pen and paper before you start coding

- Don’t worry if you don’t finish it today

Summary

- Today we've learned about
  - Pass by value
  - Scope
  - Automatic lifetime
  - If statements
  - Loops

Next time

- References and const references

- Pointers and when (not) to use them

Homework

- Read chapter 4 of the book if you have not already done so
- Finish the last exercise if you didn’t have time today
- Complete the “drill” from the end of chapter 4
- Read chapter 7 to get a head start on references

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 Uni:

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

Where to find Tom Breza:

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

My stuff:

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

See you next time! 🙂