Session 5

Tristan Brindle

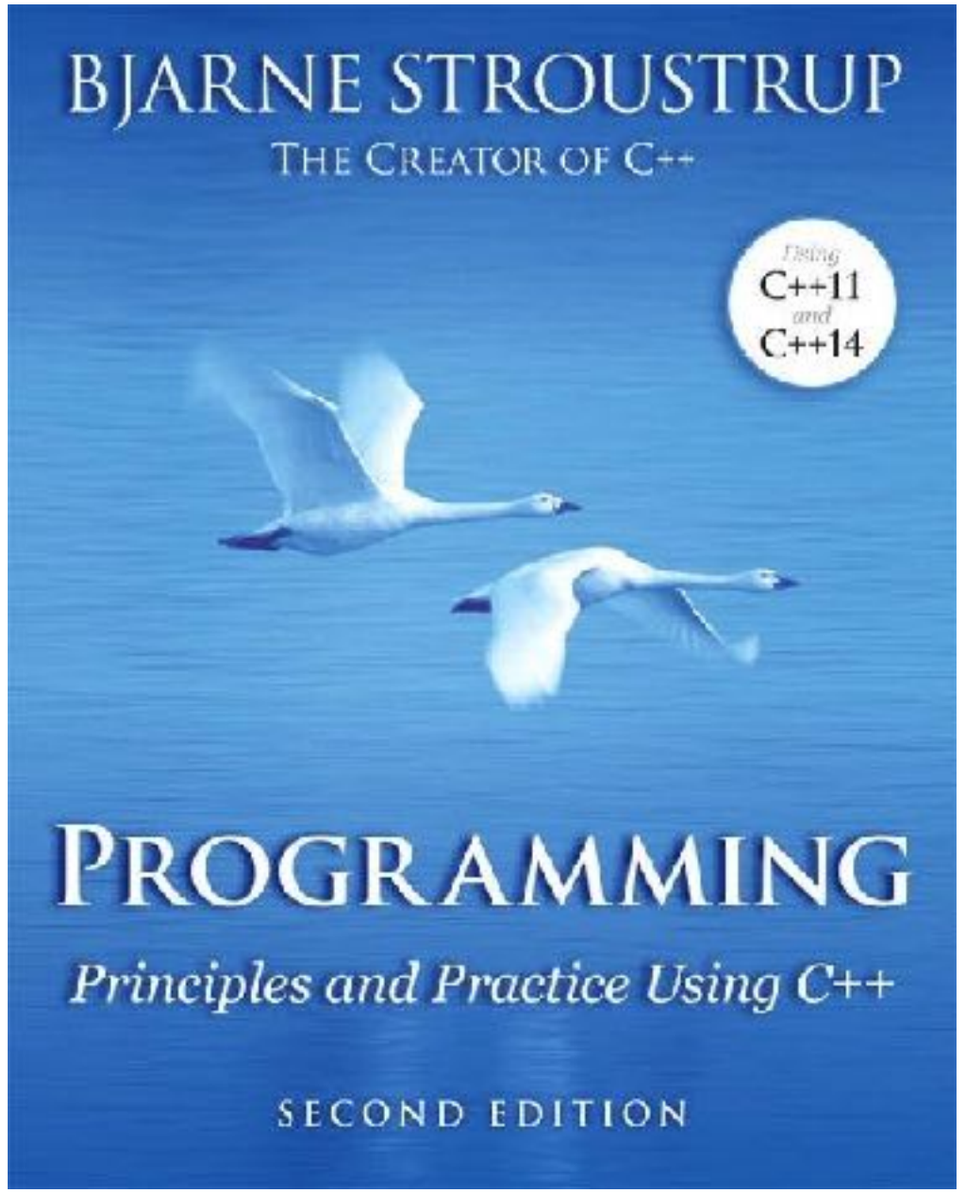
Feedback

- 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…

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

This week…

- Reference semantics
- References and const references in C++
- Introduction to pointers

Revision: value semantics

- Unlike many other programming languages, 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 (i.e. copies are “deep”).

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"
```

Reference semantics

- Value semantics are great: they allow us to *reason locally* about our functions

- However, copying large amounts of data for each function call is potentially very slow

- This is one of the reasons C++ allows us to optionally use *reference semantics* as well

- References are also needed for object-orientated programming, as we'll see later in the course

References in C++

- We can declare a reference to an existing variable by using a & after the type name, for example:

```
int i = 0;
int& ref = i;
```

- You always need to initialise a reference!
- You can think of a reference as a *new name* for an existing variable
- Anything we do to the reference affects the original variable

References example


```cpp
int i = 0;     // i is an an int
int& ref = i; // ref is a reference to int

ref = 42;  // i is now equal to 24

i = 24; // i is now equal to 24

std::cout << ref << '\n';
// prints 24

int& ref2 = ref;
// ref2 is another reference to i
```

Pass by reference

- As we saw last week, C++ uses *pass-by-value* by default for function calls

- This means that a function will receive a *copy* of the arguments you pass to it

- But we can also use *pass-by-reference*, by declaring that a function parameter has *reference type* (with &)

- This means that the function will operate *directly* on the argument you pass to it, not a copy

Pass-by-reference example


```
void say_hello(std::string& name)
{
    name = "Hello " + name;
}

std::string me = "Tristan";

say_hello(me);

std::cout << me << '\n';
// prints Hello Tristan
```

Exercise 1

- In CLion, create a new C++ executable project
- In `main.cpp`, write a function `void increment(int& i)` which adds one to `i`
- In your `main()` function, call `increment()` on a local `int` variable and print the result to see that it modifies the argument you pass to it
- Try removing the & from the definition of `increment()`. Does the code still compile? What happens? Why?
- Try calling `increment(42)`. Does the code still compile? What happens? Why?

Solution


```cpp
#include <cassert>
#include <iostream>

void increment(int& i)
{
    ++i;
}

int main()
{
    int val = 0;

    increment(val);

    assert(val == 1);

    std::cout << val << '\n'; // prints 1
}
```

Exercise 2

- Write a new function

`swap(std::string& s1, std::string& s2)`

- In this function, swap the values of `s1` and `s2` — that is, after the function returns, `s1` should contain the old value of `s2`, and `s2` should contain the old value of `s1`

- Use `cout` or `assert()` to show that your function works correctly

Solution 2


```cpp
#include <string>
#include <cassert>

void swap(std::string& s1, std::string& s2)
{
    auto temp = s1;
    s1 = s2;
    s2 = temp;
}

int main()
{
    std::string s1 = "Hello";
    std::string s2 = "World";

    swap(s1, s2);

    assert(s1 == "World" && s2 == "Hello");

}
```

Const references

- We saw in week 2 that we can declare a variable as `const`, meaning that we cannot change its value once it is initialised

- For example:

```
const int life_uni_and_everything = 42;
```

- We can also declare a reference as `const`, meaning that the value of the variable cannot be changed *via that reference*

- For example:

```
int i = 42;
const int& ref = i;
```

Const references example


```cpp
int i = 0;

const int& cref = i;

i = 24;

std::cout << cref << '\n';
// prints 24

cref += 1; // Compile error: cref is const

const int ci = 42;

int& mref = ci; // Compile error:: ci is const
```

Pass-by-const-reference

- As you might expect, we can declare a function parameter with *const reference* type

- This means that we the function will receive a reference to the passed-in variable, but cannot change its value

- This is the best of both worlds: we avoid copying, but don’t have to worry that calling a function will modify our variables

- Pass-by-const-reference is (probably) the most common way to pass arguments to functions in C++

Pass-by-const-reference example


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

int life_etc = 42;

print_int(life_etc);
// prints 42;
```

Exercise 3

- Write a function `print_string()` that takes a const reference to a string as an argument, and prints the string using `cout`

- Call your function with a local string variable. Verify that it works correctly.

- Call your function with a *string literal*. What happens? Why?

- Modify your function so that it takes the string by value instead. Does the code still compile? What happens? Why?

Solution 3


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

void print_string(const std::string& str)
{
    std::cout << str << '\n';
}

int main()
{
    std::string hello = "Hello World";

    print_string(hello);
    print_string(std::string("Hello again"));
    print_string("Hello a third time");
}
```

Pointers

“Oh no, pointers! 😩”

–Programmer before learning C++

“Oh, no pointers! 😃”

–Programmer after learning C++

Pointers

- A pointer is a value representing the *memory address* of some other variable

- We often call them “raw pointers” (as opposed to “smart pointers” which we’ll learn about later)

- Pointers are used much less frequently in C++ than in C

- Rule of thumb: *use references when you can, pointers when you have to*

Pointers

- We can obtain the *memory address* of a variable by writing an ampersand & in front of the variable's name, for example

```
int i = 0;
auto addr = &i;
```

- Here, the variable `addr` has type *pointer-to-*`int`

- We write a pointer type by putting an asterisk * after the type name

- So the above could be written as

```
int i = 0;
int* addr = &i;
```

Pointers

- Unlike references, pointers are value types
- This means we can copy them, or make them point to a different element
- Pointers are variables: that means we can take the address of a pointer
- This forms a pointer-to-pointer, written (e.g.) `int**`

Pointers

- Unlike references, pointers can be changed to point to a different memory address once initialised. For example

```
int i = 0;
int j = 1;
int* p = &i; // p contains the address of i
p = &j; // p now contains the address of j
```

Pointers

- We can *dereference* a pointer by writing a * in front of its name, for example

```
int i = 0;
int* p = &i;
*p = 4;
// i is now equal to 4
```

- When we call *p on a pointer, we are given a *reference* to the object at that memory address

- This allows us to view or modify the pointed-to object

Exercise 4

- Modify your swap() function so that it takes two pointers to strings instead, that is, its signature is

```
void swap(std::string* s1, std::string* s2)
```

- Use dereferencing operations to perform the swap as you did with references

- Use the address-of operator to call your `swap()`, passing pointers to two strings

- Use `assert()` to show that your swap works correctly

Solution 4


```cpp
#include <cassert>
#include <string>

void swap(std::string* p1, std::string* p2)
{
    auto temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main()
{
    std::string str1 = "Hello";
    std::string str2 = "World";

    swap(&str1, &str2);

    assert(str1 == "World" && str2 == "Hello");
}
```

Summary

- Today we've learned about
  - References
  - Const references
  - Passing by reference
  - Basic pointers

Homework

- Complete exercise 3 of chapter 17

- This will require reading up on pointer arithmetic, string literals and array access — you can find all this in chapter 17

- Try your exercise again, replacing `char*` with `std::string&`. What changes did you need to make to the function implementation?

- Complete exercises 4 and 5 of chapter 17

Next time

- ~~Lounging on the beach~~

- More about pointers

- Lots more exciting things

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…)