Session 9

Tristan Brindle

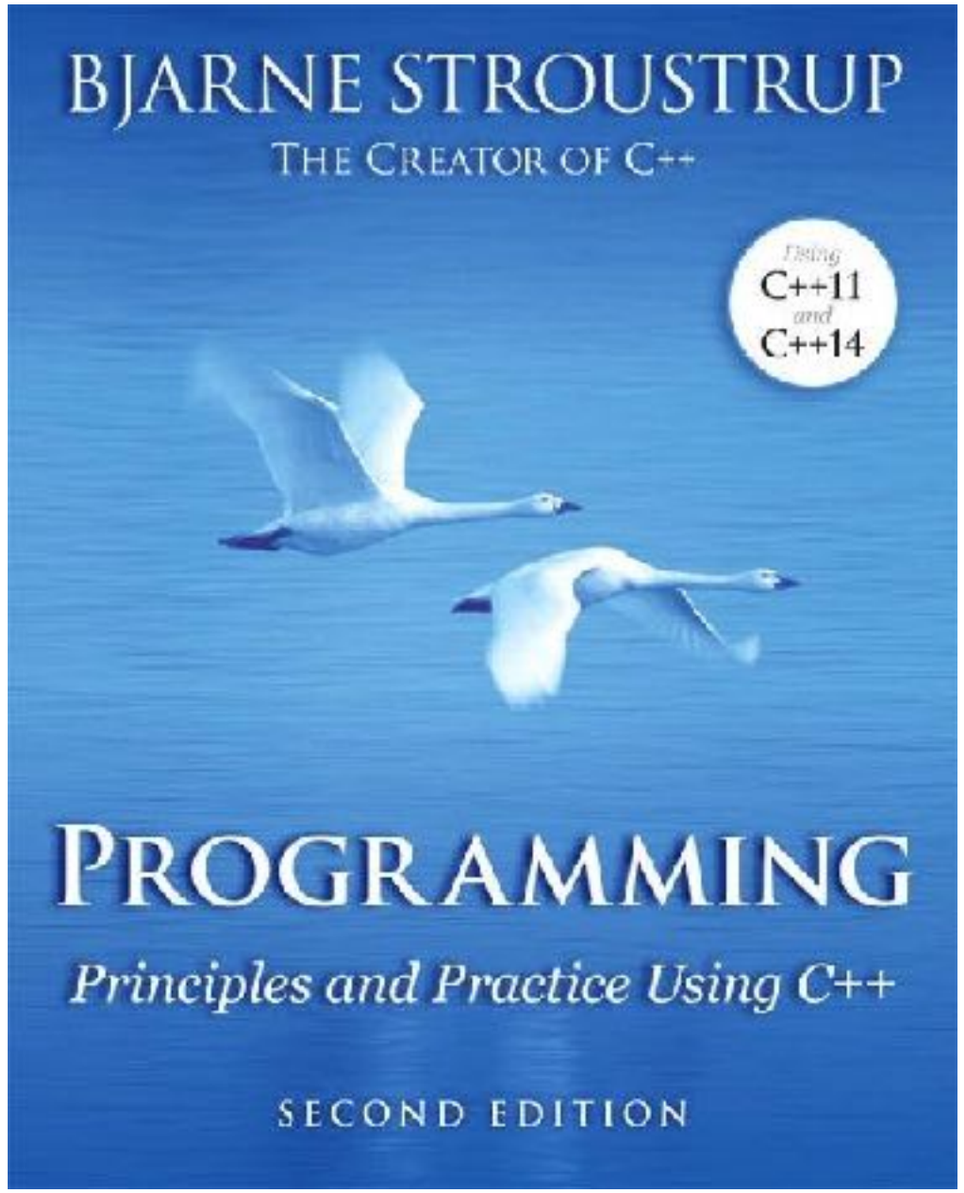
Feedback

- 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



This session

- A bit of revision
- A surprise test!
- ~~All~~ some stuff about `std::vector`

Some revision

Types

- A *type* is a way of given meaning to some data — it tells us what the data represents and what we can do with it

- For example, we can multiply two numbers, but we cannot meaningfully multiply two strings

- C++ is a *statically typed* language: every variable has its type determined at compile time

- C++ is a *strongly typed* language: we cannot change the type of a variable once declared

Functions

- Every function in C++ returns zero or one value(s). If the function does not return a value, its return type is `void`

- C++ functions can take zero or more *parameters*: each parameter has a type and an optional name

- We can also provide *default parameters* which are used if the argument is not provided

- In C++, we can provide multiple functions with the same name, but different parameter lists; this is called *overloading*

Functions

- A function must be *declared* before it can be called — the compiler must know the return type and parameters that the function takes

- Typically, we put the function declaration in a header file, and the function definition in an implementation file

Variables

- A *variable* is (roughly) a “named storage location for some data”

- In C++, every variable has a type, which dictates the kind of data it holds

- In most cases, the *lifetime* of a variable is tied to the *scope* (block) in which it is declared

- This is the basis for RAII, the central resource management technique in modern C++

Variables

- Whenever we declare a variable, we should provide an initial (default) value

- **Always initialise your variables!**

- We can use the `auto` keyword to allow the type of the variable to be deduced from its initialiser

References

- We declare a reference type by appending a & to the type name

- We can think of a reference as a new *name* for an existing variable

- They are most often use pass parameters *by reference* to functions

- Use `const` Type& function parameters to avoid copying; use plain Type& (rarely) when you need to modify a passed-in argument

Pointers

- A pointer is a value which holds a *memory address*
- We declare a pointer by saying `Type*`
- We can take the address of a variable by saying `&name`
- **Always initialise your variables!** If you don't have an appropriate value for the pointer (yet), use `nullptr`
- Dereference a pointer by writing `*name` — this returns a *reference* to the pointed-to variable
- Dereferencing an invalid pointer is *undefined behaviour* — anything could happen!

Loops

- A *while loop* runs until a given condition is `false`, running the check *before* the loop body

- A *do-while* loop runs until a given condition is `false`, running the check *after* the loop body

- A *for loop* is like a while loop that lets us declare a variable and an increment condition

- A *range-for loop* runs for every element in a compatible container

- We can break out of a loop at any time using the `break` keyword; we can jump back to the top of the loop using `continue`

Any more questions?

Eeek, it’s a test!

- Please go to
  https://www.surveymonkey.com/r/B7JBJM5
- Take your time!
- Feel free to use any resources you like

std::vector

Containers

- `std::vector` is one of the C++ standard library's *sequence containers*

- Containers are used to hold *elements* of some other type, called the *value type*

- There are several containers available in the standard library, with varying performance characteristics…

- …but `std::vector` is by far the most commonly used

Vector overview

- `std::vector` is C++'s version of a *dynamic array*
- This means that its elements are laid out sequentially in memory
- `vector` uses *dynamic allocation* to store an *arbitrary number of elements*
- `vector` provides fast random access to its elements, fast iteration, fast appending of elements, and good *locality of reference*
- TL;DR: vector is really fast 🙂

Constructing a vector

- To use `std::vector`, we first need to `#include <vector>` near the top of our source file

- To create a vector, we need to tell the compiler what kind of elements it will contain

- We do this by writing the value type in angle brackets, for example:

```
std::vector<int> vec; // vec is a vector of ints
```

- `std::vector` is an example of a *class template*: this is (superficially) similar to using generics in other languages

- In C++, the value type is part of the vector's *type*: that is, `std::vector<int>` and `std::vector<float>` are *different types*

Constructing a vector

- We can supply some elements when we construct the vector by writing them in curly braces, for example:

```
std::vector<int> vec1{1, 2, 3};
std::vector<std::string> vec2{"Hello", “world"};
```

- We can also construct a vector by providing an initial count of elements, and an optional default value, for example:

```
std::vector<int> vec1(10'000'000); // vec1 contains 10 million zeros

std::vector<std::string> vec2(10, “hello");
//vec2 contains 10 copies of “hello"
```

- To use these constructors, we need to write them with *round brackets*

Adding elements

- We can add elements to the end of the `vector` using the `push_back()` *member function*

- (Member functions are often called *methods* in other languages — we'll cover them a lot more in the coming weeks)

- We can also remove elements using `pop_back()`. For example:

```
std::vector<int> vec{1, 2, 3};
vec.push_back(99); // vec now contains [1, 2, 3, 99]
vec.pop_back(); // vec contains [1, 2, 3] again
```

Size and capacity

- We can ask a vector for its *size* — the number of elements it contains — using the `size()` member function:

```
std::vector<int> vec{1, 2, 3};
assert(vec.size() == 3);
```

- As well a its size, a vector also has a *capacity* — this is the number of elements it can store before it needs to reallocate.

Size and capacity

- We can ask the vector for its capacity using the `capacity()` member function:

```
std::vector<int> vec{};
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
std::cout << vec.capacity();
```

- We can override the capacity using the `reserve()` member function

- If we have removed elements from the vector, we can recover the unused memory using the `shrink_to_fit()` member function

Element Access

- Elements of a vector are stored in the order in which they are added
- Every element of the vector has an *index*, counted from zero
- We can access an any element by writing its index in square brackets after the vector's name, for example:

```
std::vector<int> vec{5, 4, 3, 2, 1};
assert(vec[0] == 5);
assert(vec[4] == 1);
```

- This returns a *reference* to the contained element, meaning we can use it to change the value of a stored element:

```
std::vector<int> vec{5, 4, 3, 2, 1};
vec[2] = 42;
// vec now contains [5, 4, 42, 2, 1]
```

Range access

- As a standard container, vector can be used in a range-for loop. This can be used to perform some action for every element in the vector. For example:

```
std::vector<int> vec{1, 2, 3};

for (auto i : vec) {
    std::cout << i << ' ';
}
// prints 1 2 3
```

Range access

- Remember, `auto` *never deduces to a reference*. That means if we use plain `auto` in a range-for, we will *copy* every element as we iterate through the container.
- It's almost always better to use `auto&` if you want to modify the elements, or `const auto&` if you do not. For example:

```
void fill_vector(std::vector<int>& vec)
{
    int value = 0;
    for (auto& i : vec) {
        i = value++;
    }
}
```

Further reading

- Textbook Chapter 18
- Reference documentation on cppreference:

  http://en.cppreference.com/w/cpp/container/vector

Exercise

- https://classroom.github.com/a/-j8N-u8T

Homework

- Complete the standard library vector drill from the end of chapter 18

Next time

- User-defined types

- Structs, classes and member functions

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…)