C++ London

uni

Containers and algorithms — session 1

Tristan Brindle

Feedback

C++ London uni

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

Advent of Code

- For those who like programming challenges, Advent of Code is a daily series of problems running from 1st December to Christmas Day
- Join up at adventofcode.com
- If you like, you can join the C++ London Uni leaderboard! See the Slack channel for the code.

Last week

- The end of the Custom Types module
- Public and private member access
- End-of-module quiz

This week

- End-of-module quiz! (again)
- Quiz solutions
- Introduction to the STL

Quiz time!

- http://tiny.cc/cpplondonuni2018retake

Quiz answers

- Over to Oli 🙂

Any questions before we move on?

STL motivation

- Problem: we have a vector of test scores, marked out of ten. We want to find out how many students scored a perfect 10.

- How can we do this?

```cpp
const std::vector<int> scores = get_scores();

int num_tens = 0;
for (int i : scores) {
    if (i == 10) {
        ++num_tens;
    }
}
```

STL motivation

- Problem: we have been given a list of names of people’s pet dogs. How many of these dogs are called “Rover”?

```cpp
const std::vector<std::string> names = get_dog_names();

int num_rovers = 0;
for (const auto& name : names) {
    if (name == "Rover") {
        ++num_rovers;
    }
}
```

STL motivation

- Problem: we are writing some home automation software. We have an array of six light switches, and need to know how many of these switches are in the *on* state

```
const std::array<bool, 6> switches = get_switches();

int num_active = 0;
for (bool b : switches) {
    if (b) {
        ++num_active;
    }
}
```

STL motivation

- Each of these problems has the same requirement: we need to *count* the number of elements equal to a certain value

- Our solutions involved writing the same code multiple times

- A key notion in programming is *don't repeat yourself* (DRY) — if you find yourself writing the same code many times, try abstracting it out into a function

- Wouldn't it be good if there was a library which provided a `count()` function for us?

STL motivation

```cpp
#include <nanorange.hpp>

const std::vector<int> scores = get_scores();
auto num_tens = nano::count(scores, 10);

const std::vector<std::string> names = get_dog_names();
auto num_rovers = nano::count(names, "Rover");

const std::array<bool, 6> switches = get_switches();
auto num_active = nano::count(switches, true);
```

STL motivation

- By using `nano::count` rather than a hand-written for loop, we can *reduce the amount of repetition* in our code

- We can *avoid bugs* and *improve performance* by re-using code written by domain experts

- We can also more clearly *express the intention* of our code

- The standard library provides more than 90 algorithms for us to choose from!

First, there was the STL…

- The **standard template library** (STL) was created by Alexander Stepanov with Meng Lee at Hewlett Packard

- First published in 1994, it revolutionised C++, and popularised the idea of generic programming

- The bulk of HP’s STL was incorporated into the first C++ standard library in 1998

- It is still common today (although technically incorrect) to refer to the containers and algorithms part of the standard library as “the STL”.

First, there was the STL…

- The STL provided *containers* (vectors, linked lists, associative arrays and more) and *algorithms* which operate on these containers, along with some support facilities

- Stepanov’s key insight was to use C++ templates to *decouple* algorithms from containers, with *zero overhead* in terms of memory or performance

Decoupling

- Before the STL, it was common to implement specialised algorithms for each container (see `std::string` for example)
- For N containers and M algorithms, this leads to N x M implementations
- With the STL, we can write a *generic* version of each algorithm which operates on any compatible container, as efficiently as if we had implemented it directly
- Now with N containers and M algorithms, we have N + M implementations

“STL 2.0”

- The STL has been largely unchanged since C++98

- C++20 will introduce many new facilities which make life easier — you can think of it as “STL 2.0”

- NanoRange is my implementation of the C++20 proposals

- We’ll be using it today to get started, but in the coming sessions we’ll just be using the existing C++ standard library facilities

- However, you’re welcome to continue using NanoRange for exercises and homework problems if you wish

Exercise

- `https://github.com/CPPLondonUni/stl_week1_class_exercise`

Solution

- https://github.com/CPPLondonUni/stl_week1_class_exercise/tree/solution

Next week

- More on the STL
- Pointers and iterators

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…)