C++ London Uni

Containers and algorithms — session 3

Tristan Brindle

Feedback

- 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.

- My solutions: https://github.com/tcbrindle/advent_of_code_2018

- Oli’s solutions: https://github.com/Olipro/AdventOfCode-2018

Last week

- Hardcore session introducing iterators

This week

- More about iterators
- Some fun! 🎄🎅🏼🎄🎅🏼🎄🎅🏼

Revision: iterators

- *Iterators* are the “glue” that binds together STL containers and algorithms

- Containers *provide* iterators, and algorithms *use* them

- For example:

```cpp
std::vector<int> vec{5, 4, 3, 2, 1};
auto first = std::begin(vec); // iterator to start of container
auto last = std::end(vec); // iterator to end of container
std::sort(first, last); // call algorithm on iterator pair
```

- There is no single iterator *class* — rather, an iterator is a generic *concept* (or family of concepts) which classes can *model*

Revision: iterators

- An iterator can be thought of as just an *index* into some collection
- Iterators are generally used in *pairs* — almost all standard algorithms operate on a pair of iterators
- A pair of iterators denotes a *range*
- The first iterator in the pair points to the *start of the range*
- The second iterator in the pair points to one place *past the end of the range*

Revision: iterators

- We can obtain an iterator to the start of a container by calling `container.begin()`

- We can obtain an iterator to (one past) the end of a container by calling `container.end()`

- Iterators are value types: they can be copied, assigned to, compared for equality etc

- Iterators are small, cheap to construct and cheap to copy: the STL algorithms copy them around freely

Revision: iterators

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

auto it1 = vec.begin();
// We can copy iterators
auto it2 = it1;
// it2 denotes the same position in the same collection
assert(it1 == it2);
// We can assign to iterators
it2 = vec.end();
// The iterators are no longer equal
assert(it1 != it2);
```

Dereferencing iterators

- For a valid iterator, we can obtain a reference to the element of the collection that it points to

- This is called *dereferencing* the iterator

- We write this as `*iter`

- If the returned reference is *read-only*, we call the iterator a *const iterator*.

- **It is an error to dereference an invalid iterator!**

Dereferencing iterators

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

auto it1 = vec.begin();
std::cout << *it1 << '\n';
// prints 1
*it1 = 42;
std::cout << *it1 << '\n';

const std::vector<int> cvec{3, 2, 1};
auto it2 = cvec.begin();
std::cout << *it2 << '\n';
// prints 3
*it2 = 42;
// Compile error -- it2 is a const iterator

auto it3 = vec.end();
std::cout << *it3 << '\n';
// Undefined behaviour -- it3 is not a valid iterator
// (May print junk, or just crash)
```

Incrementing iterators

- We can *increment* a valid iterator so that it points to the next position in the collection
- We write this as `++iter` (as with `int`s)
- As with `int`s, we can also write `iter++`, which increments the iterator but returns the previous position
- The standard library function `std::next(iter)` returns a new iterator which points to one place after `iter`.

Incrementing iterators

```
std::array<float, 12> arr{0.0f, };

auto it1 = arr.begin();
// it1 points to the element at position zero
++it1;
// it1 points to the element at position one
auto it2 = std::next(it1);
// it2 points to the element at position two
// it1 still points to the element at position one

// What does this do?
for (auto it = arr.begin(); it != arr.end(); ++it) {
    *it = 99;
}
```

Any questions before we move on?

Range-for loops

- A type which meets the standard library's *Container* requirements can be used in a *range-for loop*

- This means any type for which `begin()` and `end()` return types which meet the iterator requirements

- For example:

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

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

-

Iterator types

- The *type* of a container's iterator depends upon its implementation
- So a `vector` iterator is different to a `list` iterator, which is different to an `unordered_map` iterator and so on
- If necessary, you can obtain the type of a container's iterator using its nested `::iterator` type, for example:

```
std::vector<int> vec{5, 4, 3, 2, 1};
typename std::vector<int>::iterator iter = vec.begin();
```

- However, with `auto` in C++11 this is very rarely needed

Const iterators

- An iterator which provides *read-only* access to a container's elements is called a *const iterator*
- We obtain a const iterator by calling `begin()` or `end()` on a *const* instance of a container, or by calling the `cbegin()` and `cend()` functions
- A *non-const* iterator can be converted to a *const* iterator, but not vice-versa
- A const iterator means that the element pointed to is treated as const, *not the iterator itself*!

Beware iterator invalidation!

- If we hold an iterator to a container, then that iterator can become invalidated if the container's internal data structures are changed

- Such an invalidated iterator is often called a *dangling iterator*. Dangling iterators are a frequent source of bugs in C++ programs, and the compiler can do little to help.

- It is an error to dereference or advance an invalid iterator. All we can safely do is destroy it or re-initialise it via assignment

- For example, calling `push_back()` on a `std::vector` potentially reallocates the vector's internal array, invalidating all iterators to that vector

- The standard library provides details about which member functions potentially invalidate iterators. Const member functions do not invalidate, as they do not modify the container.

It’s party time! 🥳🎄🎅🏼

Next week

- It’s Christmas!

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…)