Containers and algorithms — session 4

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”

Last time

- Iterators and basic iterator operations
- Christmas party!

This week

- Iterator categories
- An overview of standard containers

Revision: iterators

- *Iterators* are the “glue” that binds together STL containers and algorithms
- Containers *provide* iterators, and algorithms *use* them
- For example:

```
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 *type* — rather, an iterator is a generic *concept* (or family of concepts) which types may *model*

Revision: iterators

- An iterator represents a **position** in a sequence
- 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

```
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);
```

Revision: iterators

- We can access the element that a valid iterator points to by writing `*iter`. This is called *dereferencing* the iterator.
- If the returned reference is *read-only*, then the iterator is called a *const iterator*
- We can *advance* a valid iterator to point to the next element in a sequence by writing `++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?

Iterator categories

- So far we have only discussed the basic iterator operations
- Some iterators offer more functionality, which can be used to implement more efficient or more complex algorithms
- The level of functionality an iterator offers is called its *category*
- Some algorithms can only be called on certain categories of iterator. For example, `std::sort()` only works with *random access iterators*
- Some algorithms provide more efficient implementations when used with higher iterator categories

Iterator categories

- The are five iterator categories, forming a hierarchy
  - Single-pass iterators (*input* and *output*)
  - *Forward* iterators
  - *Bidirectional* iterators
  - *Random-access* iterators
- Each category offers successively more functionality

Input iterators

- The most basic category is Input Iterator
- Input iterators are those whose values we can *read from*
- Input iterators are *single-pass* — once we have incremented an iterators, all copies are invalidated!
- An example of an input iterator is `std::istream_iterator`
- An example of an algorithm that operates on input iterators is `std::count()`

Output iterators

- Output iterators are those we can write through, by saying `*iter = value`

- Like input iterators, output iterators are *single-pass*

- An example of an output iterator is `std::ostream_iterator`

- Output iterators most often appear as “out parameters” in standard algorithms, for example `std::copy()`

- Iterators of higher categories which are also writable are called *mutable* iterators

Forward iterators

- Forward iterators are input iterators which we can read from multiple times (i.e. they are not *single-pass*)
- Unlike pure input iterators, it is generally okay to store a forward iterator and read from it later
- An example of a forward iterator is `std::forward_list::iterator`
- Many standard algorithms require at least forward iterators, for example `std::unique()`

Bidirectional iterators

- Bidirectional iterators are forward iterators which we can also use to traverse *backwards* through the range

- We can step a bidirectional iterator backwards by saying `--iter`

- An example of a bidirectional iterator is `std::list::iterator`

- Only a few standard algorithms require bidirectional iterators, for example `std::stable_partition()`

Random-access iterators

- A random access iterator is a bidirectional iterator which we can advance forward or backwards by an arbitrary distance *in constant time*
- (We could advance a forward iterator N places just by calling `++iter` N times, but this would be hugely inefficient for large containers!)
- Random access iterators provide `operator+()`, `operator-()`, `operator+=()` and `operator-=()` for moving arbitrary distances
- The canonical example of a container with random access iterators is `std::vector`
- A raw pointer to an element of a C array is also a random access iterator
- Random access iterators are generally required by the standard library's sorting operations, for example `std::sort()`

Any questions before we move on?

Standard containers overview

- The C++ standard library provides several container classes which can (and should!) be used when writing our own programs
- These can broadly be divided into four categories:
  - Sequence containers: elements are stored in the order they are added
  - Associative containers: keys are *sorted* for fast lookup
  - Unordered associative containers: keys are *hashed* for fast lookup
  - Container adaptors: provide a modified interface for specific tasks

Standard containers overview

- All standard containers are *class templates*
- This means that when you create an *instance* of a container to hold a particular type, it is specialised *just for that type*
- This means that a `std::vector<int>` is not the same type as `std::vector<double>`, and `std::list<std::string>` is not the same type as `std::list<bool>`
- Some containers have parameters which can be used to specialise the behaviour — for example the unordered containers can use a customised hash function
- All the standard containers have an `Allocator` template parameter which can be used to optimise memory allocations in advanced use cases

Standard containers overview

- The standard containers are designed to have a consistent programmer interface

- For example, `std::vector`, `std::list` and `std::deque` all have a `push_back()` member function which appends an element to the end of the sequence

- In particular, all the containers have `begin()` and `end()` member functions which return *iterators*

- They also have `cbegin()` and `cend()` functions which return *const iterators* — these provide read-only access to the container

- See https://en.cppreference.com/w/cpp/container for complete details

Sequence containers

- `std::array`: Fixed-size random-access array
- `std::vector`: Dynamically-sized random-access array
- `std::list`: Doubly-linked list
- `std::forward_list`: Singly-linked list
- `std::deque`: Double-ended queue
- `std::string`: Dynamically-sized random-access array of characters

`std::vector`

- `std::vector` is the standard's version of a dynamically-sized, random-access array
- Properties:
    - Element access is constant time
    - Adding elements to the end of a vector is *amortised constant time*
    - Adding elements anywhere else is *linear* in the number of elements in the vector
    - Iterators are random-access (guaranteed contiguous in C++17)
- `std::vector` should be your default, go-to container for most uses
- **When in doubt, use vector!**

`std::list`

- `std::list` is a doubly-linked list
- Properties:
    - Adding/removing an element anywhere in the list is constant time
    - Accessing a particular element is linear in the number of elements in the list
    - Accessing the `size()` of the list is constant time in C++11
    - Iterators are *bidirectional*
- Note that `std::list` is *node-based*, meaning adding an element requires a dynamic allocation
- Guideline: prefer `std::list` to `std::vector` only when frequently inserting and removing elements from the middle of a *large* sequence

`std::deque`

- `std::deque` is a *double-ended queue*
- Properties:
    - Element access is constant time
    - Insertion or removal at the end **or the beginning** of a deque is constant time
    - Insertion or removal elsewhere is linear in the size of the deque
    - Iterators are random access
- `std::deque` offers the same complexity guarantees as `std::vector`, plus an extra guarantee regarding insertion at the start
- But there's no such thing as a free lunch: `deque`'s operations have a larger constant factor, and there is likely to be extra memory overhead

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…)