C++ London University Session 21

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- Please join the #ug_uk_cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an “invitation”

Today’s Lesson Plan

- Intro to STL
- Iterators and ranges
- Exercise: writing your own iterator
- Next week: STL containers and algorithms

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

Introducing Iterators

- Iterators are the “glue” that binds together 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*

What is an iterator?

- The simple answer: an iterator is just an index into some collection
- (Trivia: Stepanov originally called them *linear coordinates*)
- The complicated answer: iterators are a *generalisation of pointers*
- Just as a raw pointer can point to an element of a C array, so an iterator points to an element of a more complex container

Iterator Fundamentals

- An iterator points to an element of a collection
- Iterators should be cheap to construct, and cheaply copyable
- We can *dereference* an iterator to access the element it points to by saying `*iter`
- We can *advance* an iterator to point to the next element of the collection by saying `++iter` (or `std::advance(iter)`)
- We can *compare iterators* for equality (that is, if they point to the same element) using `iter1 == iter2` and `iter1 != iter2`
- (This is not a complete list of standard library Iterator concept requirements, see cppreference.com's extensive documentation for the gory details)

Iterator Fundamentals Example

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

auto iter1 = std::begin(vec); // create iterator pointing to the first element
auto iter2 = iter1; // copy iter1, iter2 now also points to the first element
assert(iter1 == iter2); // the iterators are equal

++iter1; // advance iter1 one place
assert(iter1 != iter2); // the iterators are no longer equal

int i1 = *iter1; // dereference the first iterator
int i2 = *iter2; // dereference the second iterator
std::cout << i1 << ' ' << i2 << '\n'; // prints 2 1
```

Iterators and Ranges

- 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 first element in the range
- The second iterator in the pair points to one place *past the end of the range*
- **It is an error to dereference a past-the-end iterator!**

Iterators and Ranges

- 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()`

- The standard library has free functions `std::begin()` and `std::end()` which wrap the member function calls, and also work on C arrays

- Prefer using the free versions in generic code which must operate on any sort of range

Iterators and Ranges

- Typically standard algorithms will take a pair of iterators by value, and advance the first iterator until it is equal to the last, operating on each element in turn

- For example, here is an implementation of std::count:

```cpp
// Counts occurrences of `value` in the given range
template <typename Iter, typename T>
std::size_t count(Iter first, Iter last, const T& value)
{
    std::size_t counter = 0;
    while (first != last) {
        if (*first == value) {
            ++counter;
        }
        ++first;
    }
    return counter;
}

std::vector<int> vec{1, 2, 2, 2, 3, 4};
auto num_twos = count(vec.begin(), vec.end(), 2);
std::cout << num_twos << '\n'; // prints 3
```

Range-for loops

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

- This basically means any type for which `std::begin()` and `std::end()` return valid Iterators

- 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 `container.cbegin()` and `container.cend()` member functions

- Since C++14, there are also `std::cbegin()` and cend() free 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.

Iterators++?

- As a side note, there is currently work ongoing to add direct support for ranges in the C++ standard library

- At the most basic level, this means that you will be able to pass a range directly to an algorithm, without having to call `begin()` and `end()` yourself. For example:

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

- But there's much much (much) more!

- Hopefully this will be part of C++20

- In the mean time, this is available in Eric Neibler's Range-V3 library (but not for MSVC 😟)

- https://github.com/ericniebler/range-v3/

Iterator Categories

- So far we have only discussed the basic iterator interface
- In fact there are five *categories* of iterators, forming a hierarchy
- These are:
  - Input Iterators
  - Output Iterators
  - Forward Iterators
  - Bidirectional Iterators
  - Random Access Iterators
- Some algorithms can only be called on certain categories of iterator. For example, `std::sort()` only works with random access iterators
- Other algorithms provide more efficient implementations when used with higher iterator categories

Input Iterators

- The most basic category is Input Iterator
- Input iterators are those whose values we can *read from*
- Input iterators are *single-pass* — we can only read from them once!
- 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 to, by saying `*iter = value`
- Like input iterators, output iterators are *single-pass* (we can only write to them once)
- 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 (but be careful about *iterator invalidation!*)

- 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` (or `std::advance(iter, -1)`)
- 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 random access iterator is `std::vector::iterator`

- 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()`

Iterator Traits

- A valid iterator must declare which category it is by providing an `iterator_category` member typedef, which must be one of
  - std::input_iterator_tag
  - std::output_iterator_tag
  - std::forward_iterator_tag
  - std::bidirectional_iterator_tag
  - std::random_access_iterator_tag
- We can get retrieve an iterator type's category tag using `std::iterator_traits<IterType>::iterator_category`
- The tag type is mostly useful for *tag dispatch*, a way of selecting different algorithm implementations for different iterator categories
- The `iterator_traits` class can also be used to obtain other details about an iterator, such as its `value_type` (i.e. the type of the elements it points to).

Iterator Adaptors

- As well as iterators for containers, the standard library also provides a selection of iterator adaptors

- Some of the most useful are:

    - `std::reverse_iterator<Iter>`: wraps a bidirectional iterator, moving backwards through a container. Can also be accessed by a container's `rbegin()` and `rend()` methods

    - `std::back_insert_iterator<Container>`: An output iterator which calls `push_back()` on the given container when the iterator is written to. Most commonly used for `std::vector` with algorithms such as `std::copy()`

    - `std::move_iterator<Iter>`: wraps an iterator and provides an rvalue reference to the element when dereferenced

Exercise

- Today’s exercise is not for the faint of heart!
- We will be implementing an iterator type for a simplified vector class
- We’ll start off with an input iterator, and work up to random access
- https://github.com/CPPLondonUni/iterators_exercise

Next time…

- Introduction to STL part 2: containers and algorithms

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…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Twitter: @cpplondonuni
- Github: github.com/CPPLondonUni
- Reddit: reddit.com/r/CppLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #ug_uk_cpplondonuni
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂