CS100 Lecture 20

Iterators and Algorithms

Contents

- Iterators
- Algorithms

Iterators

A generalized "pointer" used for accessing elements in different containers.

Every container has its iterator, whose type is `Container::iterator`.

e.g. `std::vector<int>::iterator`, `std::forward_list<std::string>::iterator`

- `auto` comes to our rescue!

Iterators

For any container object `c`,

- `c.begin()` returns the iterator to the first element of `c`.
- `c.end()` returns the iterator to **the position following the last element** of `c` ("off-the-end", "past-the-end").

<img src="img/range-begin-end.svg", width=800>

Iterators

A pair of iterators (`b`, `e`) is often used to indicate a range `[b, e)`.

Such ranges are **left-inclusive**. Benefits:

- `e - b` is the **length** (**size**) of the range, i.e. the number of elements. There is no extra `+1` or `-1` in this expression.
- If `b == e`, the range is empty. In other words, to check whether the range is empty, we only need to do an equality test, which is easily supported by all kinds of iterators.

Iterators

Basic operations, supported by almost all kinds of iterators:

- `*it`: returns a reference to the element that `it` refers to.
- `it->mem`: equivalent to `(*it).mem`.
- `++it`, `it++`: moves `it` one step forward, so that `it` refers to the "next" element.
  - `++it` returns a reference to `it`, while `it++` returns a copy of `it` before incrementation.
- `it1 == it2`: checks whether `it1` and `it2` refer to the same position in the container.
- `it1 != it2`: equivalent to `!(it1 == it2)`.

These are supported by the iterators of all sequence containers, as well as `std::string`.

Iterators

Use the basic operations to traverse a sequence container:

```
void swapcase(std::string &str) {
  for (auto it = str.begin(); it != str.end(); ++it) {
    if (std::islower(*it))
      *it = std::toupper(*it);
    else if (std::isupper(*it))
      *it = std::tolower(*it);
  }
}
void print(const std::vector<int> &vec) {
  for (auto it = vec.begin(); it != vec.end(); ++it)
    std::cout << *it << ' ';
}
```

Iterators

Built-in pointers are also iterators: They are the iterator for built-in arrays.

For an array `Type a[N]`:

- The "begin" iterator is `a`.
- The "end" (off-the-end) iterator is `a + N`.

The standard library functions `std::begin(c)` and `std::end(c)` (defined in `<iterator>` and many other header files):

- return `c.begin()` and `c.end()` if `c` is a container;
- return `c` and `c + N` if `c` is an array of length `N`.

Range-for demystified

The range-based for loop

```
for (@declaration : container)
  @loop_body
```

is equivalent to

```
{
  auto b = std::begin(container);
  auto e = std::end(container);
  for (; b != e; ++b) {
    @declaration = *b;
    @loop_body
  }
}
```

Iterators: dereferenceable

Like pointers, an iterator can be dereferenced (`*it`) only when it refers to an existing element. (**"dereferenceable"**)

- `*v.end()` is undefined behavior.
- `++it` is undefined behavior if `it` is not dereferenceable. In other words, moving an iterator out of the range `[begin, off_the_end]` is undefined behavior.

Iterators: invalidation

```
Type *storage = new Type[n];
Type *iter = storage;
delete[] storage;
// Now `iter` does not refer to any existing element.
```

Some operations on some containers will **invalidate** some iterators:

- make these iterators not refer to any existing element.

For example:

- `push_back(x)` on a `std::vector` may cause the reallocation of storage. All iterators obtained previously are invalidated.
- `pop_back()` on a `std::vector` will invalidate the iterators that points to the deleted element.

Never use invalidated iterators or references!

```cpp
void foo(std::vector<int> &vec) {
  auto it = vec.begin();
  while (some_condition(vec))
    vec.push_back(*it++); // Undefined behavior.
}
```

After several calls to `push_back`, `vec` may reallocate a larger chunk of memory to store its elements. This will invalidate all pointers, references and iterators that point to somewhere in the previous memory block.

More operations on iterators

The iterators of containers that support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2` are **ForwardIterators**.

BidirectionalIterator: a ForwardIterator that can be moved in both directions

- supports `--it` and `it--`.

RandomAccessIterator: a BidirectionalIterator that can be moved to any position in constant time.

- supports `it + n`, `n + it`, `it - n`, `it += n`, `it -= n` for an integer `n`.
- supports `it[n]`, equivalent to `*(it + n)`.
- supports `it1 - it2`, returns the **distance** of two iterators.
- supports `<`, `<=`, `>`, `>=`.

More operations on iterators

The iterators of containers that support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2` are **ForwardIterators**.

BidirectionalIterator: a ForwardIterator that can be moved in both directions

- supports `--it` and `it--`.

RandomAccessIterator: a BidirectionalIterator that can be moved to any position in constant time.

- supports `it + n`, `n + it`, `it - n`, `it += n`, `it -= n`, `it[n]`, `it1 - it2`, `<`, `<=`, `>`, `>=`.
- `std::string::iterator` and `std::vector<T>::iterator` are in this category.

Which category is the built-in pointer in?

More operations on iterators

The iterators of containers that support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2` are **ForwardIterators**.

BidirectionalIterator: a ForwardIterator that can be moved in both directions

- supports `--it` and `it--`.

RandomAccessIterator: a BidirectionalIterator that can be moved to any position in constant time.

- supports `it + n`, `n + it`, `it - n`, `it += n`, `it -= n`, `it[n]`, `it1 - it2`, `<`, `<=`, `>`, `>=`.
- `std::string::iterator` and `std::vector<T>::iterator` are in this category.

Which category is the built-in pointer in? - RandomAccessIterator.

Initialization from iterator range

`std::string`, `std::vector`, as well as other standard library containers, support the initialization from an iterator range:

```cpp
std::vector<char> v = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'};
std::vector v2(v.begin() + 2, v.end() - 3);  // {'c', 'd', 'e', 'f'}
std::string s(v.begin(), v.end()); // "abcdefghi"
```

Algorithms

Algorithms

Full list of standard library algorithms can be found here.

No one can remember all of them, but some are quite commonly used.

Algorithms: interfaces

Parameters: The STL algorithms accept pairs of iterators to represent "ranges":

```cpp
int a[N], b[N]; std::vector<int> v;
std::sort(a, a + N);
std::sort(v.begin(), v.end());
std::copy(a, a + N, b); // copies elements in [a, a+N) to [b, b+N)
std::sort(v.begin(), v.begin() + 10); // Only the first 10 elements are sorted.
```

Since C++20, `std::ranges::xxx` can be used, which has more modern interfaces

```cpp
std::ranges::sort(a);
std::ranges::copy(a, b);
```

Algorithms: interfaces

Parameters: The algorithms suffixed `_n` use **a beginning iterator** `begin` **and an integer** `n` **to represent a range** `[begin, begin + n)`.

Example: Use STL algorithms to rewrite the constructors of `Dynarray`:

```cpp
Dynarray::Dynarray(const int *begin, const int *end)
    : m_storage{new int[end - begin]}, m_length(end - begin) {
  std::copy(begin, end, m_storage);
}
Dynarray::Dynarray(const Dynarray &other)
    : m_storage{new int[other.size()]}, m_length{other.size()} {
  std::copy_n(other.m_storage, other.size(), m_storage);
}
Dynarray::Dynarray(std::size_t n, int x = 0)
    : m_storage{new int[n]}, m_length{n} {
  std::fill_n(m_storage, m_length, x);
}
```

Algorithms: interfaces

Return values: "Position" is typically represented by an iterator. For example:

```cpp
std::vector<int> v = someValues();
auto pos = std::find(v.begin(), v.end(), 42);
assert(*pos == 42);
auto maxPos = std::max_element(v.begin(), v.end());
```

- `pos` is an **iterator** pointing to the first occurrence of `42` in `v`.
- `maxPos` is an **iterator** pointing to the max element in `v`.

"Not found" / "No such element" is often indicated by returning `end`.

```cpp
if (std::find(v.begin(), v.end(), something) != v.end()) {
  // ...
}
```

`if`: new syntax in C++17

"Not found" / "No such element" is often indicated by returning `end`.

```cpp
if (std::find(v.begin(), v.end(), something) != v.end()) { /* (*) */ }
```

If we want to use the returned iterator in (*):

```cpp
if (auto pos = std::find(v.begin(), v.end(), something); pos != v.end())
  std::cout << *pos << '\n';
```

The new syntax of `if` in C++17: `if (init_expr; condition)`.

- `init_expr` is just like the first part of the `for` statement.
- The scope of the variable declared in `init_expr` is within this `if` statement (containing the `else` clause, if present).

Algorithms: requirements

An algorithm may have **requirements** on

- the iterator categories of the passed-in iterators, and
- the type of elements that the iterators point to.

Typically, `std::sort` requires *RandomAccessIterator*s, while `std::copy` allows any *InputIterator*s.

Typically, all algorithms that need to compare elements rely only upon `operator<` and `operator==` of the elements.

- You don't have to define all the six comparison operators of `X` in order to `sort` a `vector<X>`. `sort` only requires `operator<`.

Algorithms

Since we pass **iterators** instead of **containers** to algorithms, **the standard library algorithms never modify the length of the containers**.

- STL algorithms never insert or delete elements in the containers (unless the iterator passed to them is some special *iterator adapter*).

For example: `std::copy` only **copies** elements, instead of inserting elements.

```cpp
std::vector<int> a = someValues();
std::vector<int> b(a.size());
std::vector<int> c{};
std::copy(a.begin(), a.end(), b.begin()); // OK
std::copy(a.begin(), a.end(), c.begin()); // Undefined behavior!
```

Some common algorithms (`<algorithm>`)

Non-modifying sequence operations:

- `count(begin, end, x)`, `find(begin, end, x)`, `find_end(begin, end, x)`, `find_first_of(begin, end, x)`, `search(begin, end, pattern_begin, pattern_end)`

Modifying sequence operations:

- `copy(begin, end, dest)`, `fill(begin, end, x)`, `reverse(begin, end)`, ...
- `unique(begin, end)`: drop duplicate elements.
  - requires the elements in the range `[begin, end)` to be **sorted** (in ascending order by default).
  - **It does not remove any elements!** Instead, it moves all the duplicated elements to the end of the sequence, and returns an iterator `pos`, so that `[begin, pos)` has no duplicate elements.

Some common algorithms (`<algorithm>`)

Example: `unique`

```cpp
std::vector v{1, 1, 2, 2, 2, 3, 5};
auto pos = std::unique(v.begin(), v.end());
// Now [v.begin(), pos) contains {1, 2, 3, 5}.
// [pos, v.end()) has the values {1, 2, 2}, but the exact order is not known.
v.erase(pos, v.end()); // Typical use with the container's `erase` operation
// Now v becomes {1, 2, 3, 5}.
```

`unique` does not remove the duplicate elements! To remove them, use the container's `erase` operation.

Some common algorithms (`<algorithm>`)

Partitioning, sorting and merging algorithms:

- `partition`, `is_partitioned`, `stable_partition`
- `sort`, `is_sorted`, `stable_sort`
- `nth_element`
- `merge`, `inplace_merge`

Binary search on sorted ranges:

- `lower_bound`, `upper_bound`, `binary_search`, `equal_range`

Heap algorithms:

- `is_heap`, `make_heap`, `push_heap`, `pop_heap`, `sort_heap`

Learn the underlying algorithms and data structures of these functions in CS101!

Some common algorithms

Min/Max and comparison algorithms: (`<algorithm>`)

- `min_element(begin, end)`, `max_element(begin, end)`, `minmax_element(begin, end)`
- `equal(begin1, end1, begin2)`, `equal(begin1, end1, begin2, end2)`
- `lexicographical_compare(begin1, end1, begin2, end2)`

Numeric operations: (`<numeric>`)

- `accumulate(begin, end, initValue)`: Sum of elements in `[begin, end)`, with initial value `initValue`.
  - `accumulate(v.begin(), v.end(), 0)` returns the sum of elements in `v`.
- `inner_product(begin1, end1, begin2, initValue)`: Inner product of two vectors $\mathbf{a}^T\mathbf{b}$, added with the initial value `initValue`.

Predicates

Consider the `Point2d` class:

```
struct Point2d {
  double x, y;
};
std::vector<Point2d> points = someValues();
```

Suppose we want to sort `points` in ascending order of the `x` coordinate.

- `std::sort` requires `operator<` in order to compare the elements,
- but it is not recommended to overload `operator<` here! (What if we want to sort some `Point2d`s in another way?)

(C++20 modern way: `std::ranges::sort(points, {}, &Point2d::x);`)

Predicates

`std::sort` has another version that accepts another argument `cmp`:

```
bool cmp_by_x(const Point2d &lhs, const Point2d &rhs) {
  return lhs.x < rhs.x;
}
std::sort(points.begin(), points.end(), cmp_by_x);
```

`sort(begin, end, cmp)`

- `cmp` is a **Callable** object. When called, it accepts two arguments whose type is the same as the element type, and returns `bool`.
- `std::sort` will use `cmp(x, y)` instead of `x < y` to compare elements.
- After sorting, `cmp(v[i], v[i + 1])` is true for every `i` $\in$ `[0, v.size()-1)`.

Predicates

To sort numbers in reverse (descending) order:

```
bool greater_than(int a, int b) { return a > b; }
std::sort(v.begin(), v.end(), greater_than);
```

To sort them in ascending order of absolute values:

```
bool abs_less(int a, int b) { return std::abs(a) < std::abs(b); } // <cmath>
std::sort(v.begin(), v.end(), abs_less);
```

Predicates

Many algorithms accept a Callable object. For example, `find_if(begin, end, pred)` finds the first element in `[begin, end)` such that `pred(element)` is true.

```cpp
bool less_than_10(int x) {
  return x < 10;
}
std::vector<int> v = someValues();
auto pos = std::find_if(v.begin(), v.end(), less_than_10);
```

`for_each(begin, end, operation)` performs `operation(element)` for each element in the range `[begin, end)`.

```cpp
void print_int(int x) { std::cout << x << ' '; }
std::for_each(v.begin(), v.end(), print_int);
```

Predicates

Many algorithms accept a Callable object. For example, `find_if(begin, end, pred)` finds the first element in `[begin, end)` such that `pred(element)` is true.

What if we want to find the first element less than `k`, where `k` is determined at run-time?

Predicates

What if we want to find the first element less than `k`, where `k` is determined at run-time?

```cpp
struct LessThan {
  int k_;
  LessThan(int k) : k_{k} {}
  bool operator()(int x) const {
    return x < k_;
  }
};
auto pos = std::find_if(v.begin(), v.end(), LessThan(k));
```

- `LessThan(k)` constructs an object of type `LessThan`, with the member `k_` initialized to `k`.
- This object has an `operator()` overloaded: **the function-call operator**.
    - `LessThan(k)(x)` is equivalent to `LessThan(k).operator()(x)`, which is `x < k`.

Function objects

Modern way:

```cpp
struct LessThan {
  int k_; // No constructor is needed, and k_ is public.
  bool operator()(int x) const { return x < k_; }
};
auto pos = std::find_if(v.begin(), v.end(), LessThan{k}); // {} instead of ()
```

A **function object** (aka "functor") is an object `fo` with `operator()` overloaded.

- `fo(arg1, arg2, ...)` is equivalent to `fo.operator()(arg1, arg2, ...)`. Any number of arguments is allowed.

Function objects

Exercise: use a function object to compare integers by their absolute values.

```cpp
struct AbsCmp {
  bool operator()(int a, int b) const {
    return std::abs(a) < std::abs(b);
  }
};
std::sort(v.begin(), v.end(), AbsCmp{});
```

Lambda expressions

Defining a function or a function object is not good enough:

- These functions or function objects are almost used only once, but
- too many lines of code is needed, and
- you have to add names to the global scope.

Is there a way to define an **unnamed**, immediate callable object?

Lambda expressions

To sort by comparing absolute values:

```
std::sort(v.begin(), v.end(),
          [](int a, int b) -> bool { return std::abs(a) < std::abs(b); });
```

To sort in reverse order:

```
std::sort(v.begin(), v.end(),
          [](int a, int b) -> bool { return a > b; });
```

To find the first element less than `k`:

```
auto pos = std::find_if(v.begin(), v.end(),
                        [k](int x) -> bool { return x < k; });
```

Lambda expressions

The return type can be omitted and deduced by the compiler.

```
std::sort(v.begin(), v.end(),
          [](int a, int b) { return std::abs(a) < std::abs(b); });
```

```
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
```

```
auto pos = std::find_if(v.begin(), v.end(), [k](int x) { return x < k; });
```

Lambda expressions

A lambda expression has the following syntax:

```
[capture_list](params) -> return_type { function_body }
```

The compiler will generate a function object according to it.

```
int k = 42;
auto f = [k](int x) -> bool { return x < k; };
bool b1 = f(10); // true
bool b2 = f(100); // false
```

Lambda expressions

```
[capture_list](params) -> return_type { function_body }
```

It is allowed to write complex statements in `function_body`, just as in a function.

```cpp
struct Point2d { double x, y; };
std::vector<Point2d> points = somePoints();
// prints the l2-norm of every point
std::for_each(points.begin(), points.end(),
              [](const Point2d &p) {
                auto norm = std::sqrt(p.x * p.x + p.y * p.y);
                std::cout << norm << std::endl;
              });
```

Lambda expressions: capture

To capture more variables:

```cpp
auto pos = std::find_if(v.begin(), v.end(),
                    [lower, upper](int x) { return lower <= x && x <= upper;});
```

To capture by reference (so that copy is avoided)

```cpp
std::string str = someString();
std::vector<std::string> wordList;
// finds the first string that is lexicographically greater than `str`,
// but shorter than `str`.
auto pos = std::find_if(wordList.begin(), wordList.end(),
     [&str](const std::string &s) { return s > str && s.size() < str.size();});
```

Here `&str` indicates that `str` is captured by referece. `&` **here is not the address-of operator!**

More on lambda expressions

- *C++ Primer* Section 10.3
- *Effective Modern C++* Chapter 6 (Item 31-34)

Note that *C++ Primer (5th edition)* is based on C++11 and *Effective Modern C++* is based on C++14. Lambda expressions are evolving at a very fast pace in modern C++, with many new things added and many limitations removed.

More fancy ways of writing lambda expressions are not covered in CS100.

Back to algorithms

So many things in the algorithm library! How can we remember them?

- Remember the **conventions**:
  - No insertion/deletion of elements
  - Iterator range `[begin, end)`
  - Functions named with the suffix `_n` uses `[begin, begin + n)`
  - Pass functions, function objects, and lambdas for customized operations
  - Functions named with the suffix `_if` requires a boolean predicate
- Remember the common ones: `copy`, `find`, `for_each`, `sort`, ...
- Look them up in cppreference before use.

Summary

Iterators

- A generalized "pointer" used for accessing elements in different containers.
- Iterator range: a left-inclusive interval `[b, e)`.
- `c.begin()`, `c.end()`
- Basic operations: `*it`, `it->mem`, `++it`, `it++`, `it1 == it2`, `it1 != it2`.
- Range-based `for` loops are in fact traversal using iterators.
- More operations: BidirectionalIterator supports `it--` and `--it`. RandomAccessIterator supports all pointer arithmetics.
- Initialization of standard library containers from an iterator range.

Summary

Algorithms

- Normal functions accept iterator range `[b, e)`. Functions with `_n` accept an iterator and an integer, representing the range `[begin, begin + n)`.
- Position is represented by an iterator.
- STL algorithms never insert or delete elements in the containers.
- Some algorithms accept a predicate argument, which is a callable object. It can be a function, a pointer to function, an object of some type that has an overloaded `operator()`, or a lambda.
- Lambda: `[capture_list][params] -> return_type { function_body }`