

CS100 Lecture 22

Standard Template Library I

Contents

- STL
- Iterators
- Algorithms

Standard Template Library

Added into C++ in 1994.

- Containers
- Iterators (This lecture)
- Algorithms (This lecture)
- Function objects
- Adapters
- Allocators

Standard Template Library

Containers (e.g., `std::vector`) are used to store data.

Iterators are used to access data in **containers**.

Algorithms (e.g., `sort`) are used to manipulate data in **containers** (through **iterators**).

Iterators

Iterators

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

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

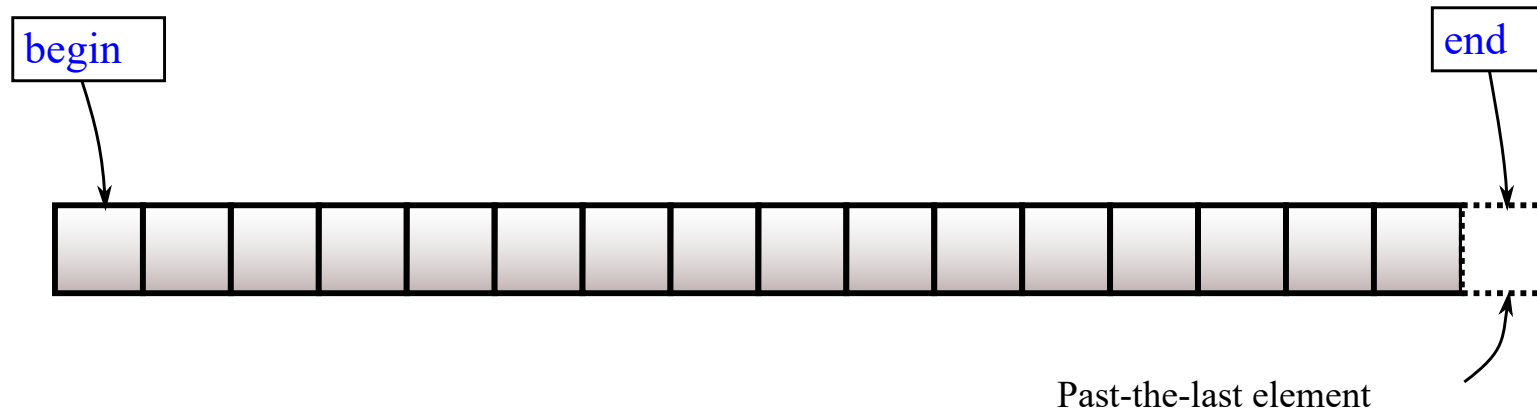
e.g., `std::vector<int>::iterator`, `std::list<std::string>::iterator`

- `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").



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 object;
- 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 is **dereferenceable** (`*it`) only when it refers to an existing element.

- `*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;  
// `iter` is invalidated. 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<T>` may cause the reallocation of storage. All iterators obtained previously are invalidated.
- `pop_back()` on a `std::vector<T>` will invalidate the iterators that points to the deleted element.

Never use invalidated iterators!

```
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 the iterators that point to somewhere in the previous memory block.

Types of iterators

Forward iterator: can move in the forward direction only.

- support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2`.

Bidirectional iterator: a forward iterator that can be moved in both directions

- support `--it` and `it--`.

Random access iterator: a bidirectional iterator that can be moved to any position in constant time.

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

Types of iterators

Forward iterator: can move in the forward direction only.

- support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2`.

Bidirectional iterator: a forward iterator that can be moved in both directions

- support `--it` and `it--`.

Random access iterator: a bidirectional iterator that can be moved to any position in constant time.

- support `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 type.

Which type is the built-in pointers in?

Types of iterators

Forward iterator: can move in the forward direction only.

- support `*it`, `it->mem`, `++it`, `it++`, `it1 == it2` and `it1 != it2`.

Bidirectional iterator: a forward iterator that can be moved in both directions

- support `--it` and `it--`.

Random access iterator: a bidirectional iterator that can be moved to any position in constant time.

- support `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 type.

Which type is the built-in pointers in? - Random access iterator.

Initialization from iterator range

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

```
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 STL algorithms can be found [here](#).

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

Algorithms: interfaces

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

```
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); // copy elements in [a, a+N) to [b, b+N)  
std::sort(v.begin(), v.begin() + 10); // only the first 10 elements are sorted.
```

Algorithms: interfaces

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

```
int a[N], b[N];  
std::copy(a, a + N, b); // copy elements in [a, a+N) to [b, b+N)  
std::copy_n(a, N, b); // copy elements in [a, a+N) to [b, b+N)
```

Algorithms: interfaces

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

```
std::vector<int> v = someValues();  
auto pos = std::find(v.begin(), v.end(), 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`.

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


Algorithms: requirements

An algorithm may have **requirements** on

- the types of the passed-in iterators, and/or
- the type of elements that the iterators refer to.

Typically, `std::sort` requires *random access iterators*, while `std::copy` allows any *forward iterators*.

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

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

Algorithms

Since we pass **iterators** instead of **containers** to the algorithms, the algorithms typically don't modify the length of the containers.

- The algorithms typically don't 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.

```
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)`,
`search(begin, end, pattern_begin, pattern_end)`

Modifying sequence operations:

- `copy(begin, end, dest_begin)`, `fill(begin, end, x)`, `reverse(begin, end)`, ...
- `unique(begin, end)` : drop duplicate elements.
 - It requires the elements in the range `[begin, end)` to be **sorted**.
 - **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`

```
std::vector v{1, 1, 2, 2, 2, 3, 5};  
auto pos = std::unique(v.begin(), v.end());  
// Now v becomes {1, 2, 3, 5, 1, 2, 2}  
// [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 data structures and algorithms 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:

```
class Point2d {  
public:  
    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<` for `Point2d` here! (What if we want to sort some `Point2d` s in another way?)

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 predicate (a **callable** object that can be called like a function). When called, it accepts two arguments whose type is the element type, and returns `bool`.
- `std::sort` will use `cmp(x, y)` instead of `x < y` to compare elements.

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, cond)` finds the first element in `[begin, end)` such that `cond(element)` is true.

```
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)`.

```
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, cond)` finds the first element in `[begin, end)` such that `cond(element)` is true.

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

Predicates

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

```
class LessThan {  
    int k_;  
public:  
    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 a `LessThan` object, with the member `k_` initialized to `k`.
- This object is **callable** since `LessThan` overloads the **function-call operator** `()`.
 - `LessThan(k)(x)` is equivalent to `LessThan(k).operator()(x)`, which is `x < k`.

Function objects

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

A **function object** (aka "functor") is an object `fo` of a class type that has `operator()` overloaded.

- `fo(arg1, arg2, ...)` is equivalent to `fo.operator()(arg1, arg2, ...)`.

Function objects

Use a function object to compare integers by their absolute values.

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

Lambda expressions

To create a callable object for an algorithm, defining a function or a class with overloaded `operator()` is not good enough:

- These functions or classes are almost used only once, but
- too many lines of code is needed, and
- you have to add the 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 }
```

- `capture list` captures external variables to be used in `function_body`.

The compiler will generate an unnamed callable 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.

```
class Point2d {  
public:  
    double x, y;  
};  
std::vector<Point2d> points = somePoints();  
// print 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 external variables:

```
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):

```
std::string str = someString();  
std::vector<std::string> wordList;  
// find the first string that is lexicographically greater than `str`,  
// and 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 reference. `&` 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.

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 callable objects (obtained through 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.
- Different types of iterators: forward iterator, bidirectional iterator (support `it--` and `--it`), random access iterator (support all pointer arithmetics).
- Initialization of STL containers from an iterator range.

Summary

Algorithms

- Normal functions accept iterator range `[b, e)`. Functions with `_n` accept an iterator `begin` and an integer `n`, representing the range `[begin, begin + n)`.
- Returned position is represented by an iterator.
- STL algorithms typically don't insert or delete elements in the containers.
- Some algorithms accept a predicate argument, which is a callable object. It can be obtained through a function, a function object, or a lambda.
- Lambda: `[capture_list][params] -> return_type { function_body }`.