

# CS100 Lecture 22

# Contents

## Standard Template Library (STL)

- Overview
- Sequence containers and iterators
- Algorithms and function objects (aka "functors")
- Associative containers

# Overview of STL

# Standard Template Library

Added into C++ in 1994.

- Containers
- Iterators
- Algorithms
- Function objects
- Some other adapters, like container adapters and iterator adapters
- Allocators

# Containers

- Sequence containers
  - `vector`, `list`, `deque`, `array` (since C++11), `forward_list` (since C++11)
- Associative containers
  - `set`, `map`, `multiset`, `multimap` (often implemented with *binary search trees*)
- Unordered associative containers (since C++11)
  - `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap` (implemented with *hash tables*)
- Container adapters: provide a different interface for sequential containers, but they are not containers themselves.
  - `stack`, `queue`, `priority_queue`
  - (since C++23) `flat_set`, `flat_map`, `flat_multiset`, `flat_multimap`

# Iterators

## Without iterators:

- Traverse an array

```
for (int i = 0; i != sizeof(a) / sizeof(a[0]); ++i)  
    do_something(a[i]);
```

- Traverse a `vector`

```
for (std::size_t i = 0; i != v.size(); ++i)  
    do_something(v[i]);
```

- Traverse a linked-list?

```
for (ListNode *p = l.head(); p; p = p->next)  
    do_something(p->data);
```

# Iterators

A generalization of pointers, used to access elements in different containers **in a uniform manner**.

## With iterators:

The following works no matter whether `c` is an array, a `std::string`, or any container.

```
for (auto it = std::begin(c); it != std::end(c); ++it)
    do_something(*it);
```

## Equivalent way: range-based for loops

```
for (auto &x : c) do_something(x);
```

# Algorithms

The algorithms library defines functions for a variety of purposes:

- searching, sorting, counting, manipulating, ...

Examples:

```
// assign every element in `a` with the value `x`.
std::fill(a.begin(), a.end(), x);
// sort the elements in `b` in ascending order.
std::sort(b.begin(), b.end());
// find the first element in `b` that is equal to `x`.
auto pos = std::find(b.begin(), b.end(), x);
// reverse the elements in `c`.
std::reverse(c.begin(), c.end());
```



# Algorithms

Example: Map every number in `data` to its rank. (“离散化”)

```
auto remap(const std::vector<int> &data) {  
    auto tmp = data;  
    std::sort(tmp.begin(), tmp.end()); // sort  
    auto pos = std::unique(tmp.begin(), tmp.end()); // drop duplicates  
    auto ret = data;  
    for (auto &x : ret)  
        x = std::lower_bound(tmp.begin(), pos, x) - tmp.begin(); // binary search  
    return ret;  
}
```

# Function objects

Things that look like "functions": *Callable*

- functions, and also function pointers
- objects of a class type that has an overloaded `operator()` (the function-call operator)
- lambda expressions

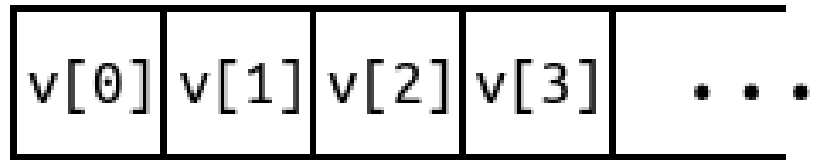
More in later lectures ...

# Sequence containers and iterators

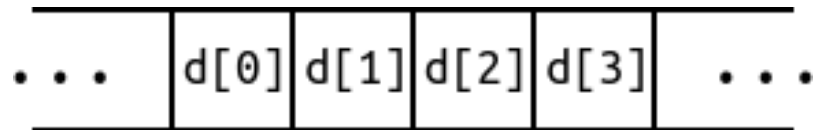
Note: `string` is not treated as a container but behaves much like one.

# Sequence containers

- `std::vector<T>` : dynamic contiguous array (we are quite familiar with)



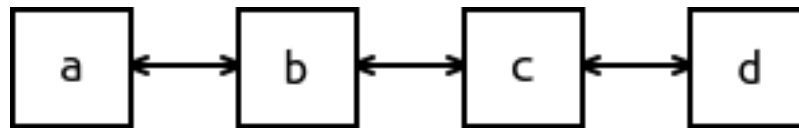
- `std::deque<T>` : double-ended queue (often pronounced as "deck")
  - `std::deque<T>` supports fast insertion and deletion **at both its beginning and its end**. ( `push_front` , `pop_front` , `push_back` , `pop_back` )



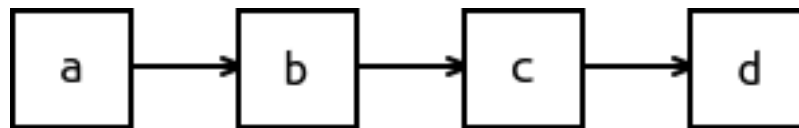
- `std::array<T, N>` : same as `T[N]` , it is a **container**
  - It will never decay to `T *` .
  - Container interfaces are provided: `.at(i)` , `.front()` , `.back()` , `.size()` , ..., as well as iterators.

# Sequence containers

- `std::list<T>` : doubly-linked list
  - `std::list<T>` supports fast insertion and deletion **anywhere in the container**,
  - but fast random access is not supported (i.e. no `operator[]` ).
  - Bidirectional traversal is supported.



- `std::forward_list<T>` : singly-linked list
  - Intended to save time and space (compared to `std::list` ).
  - Only forward traversal is supported.



# Interfaces

STL containers have consistent interfaces. See [here](#) for a full list.

Element access:

- `c.at(i)` , `c[i]` : access the element indexed `i` . `at` performs bounds checking, and throws `std::out_of_range` if `i` exceeds the valid range.
- `c.front()` , `c.back()` : access the front/back element.

# Interfaces

Size and capacity: `c.size()` and `c.empty()` are what we already know.

- `c.resize(n)` , `c.resize(n, x)` : adjust the container to be with exactly `n` elements. If `n > c.size()` , `n - c.size()` elements will be appended.
  - `c.resize(n)` : Appended elements are **value-initialized**.
  - `c.resize(n, x)` : Appended elements are copies of `x` .
- `c.capacity()` , `c.reserve(n)` , `c.shrink_to_fit()` : only for `string` and `vector` .
  - `c.capacity()` returns the capacity (number of elements that *can* be stored in the current storage)
  - `c.reserve(n)` : reserves space for at least `n` elements.
  - `c.shrink_to_fit()` : requests to remove the unused capacity, so that `c.capacity() == c.size()` .

# Interfaces

Modifiers:

- `c.push_back(x)` , `c.emplace_back(args...)` , `c.pop_back()` : insert/delete elements at the end of the container.
- `c.push_front(x)` , `c.emplace_front(args...)` , `c.pop_front()` : insert/delete elements at the beginning of the container.
- `c.clear()` removes all the elements in `c`.



# Interfaces

Modifiers:

- `c.insert(...)`, `c.emplace(...)`, `c.erase(...)` : insert/delete elements at a specified location.
  - **Warning:** For containers that need to maintain contiguous storage ( `string`, `vector`, `deque` ), insertion and deletion somewhere in the middle can be **very slow** ( $O(n)$ ).
  - These functions have a lot of overloads. Remember a few common ones, and STFW (Search The Friendly Web) when you need to use them.

# Interfaces

Some of these member functions are not supported on some containers, **depending on the underlying data structure**. For example:

- Any operation that modifies the length of the container is not allowed for `array`.
- `push_front`, `emplace_front` and `pop_front` are not supported on `string`, `vector` and `array`.
- `size` is not supported on `forward_list` in order to save time and space.
- `operator[]` and `at` are not supported on linked-lists.

[This table](#) tells you everything.

# Iterators

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

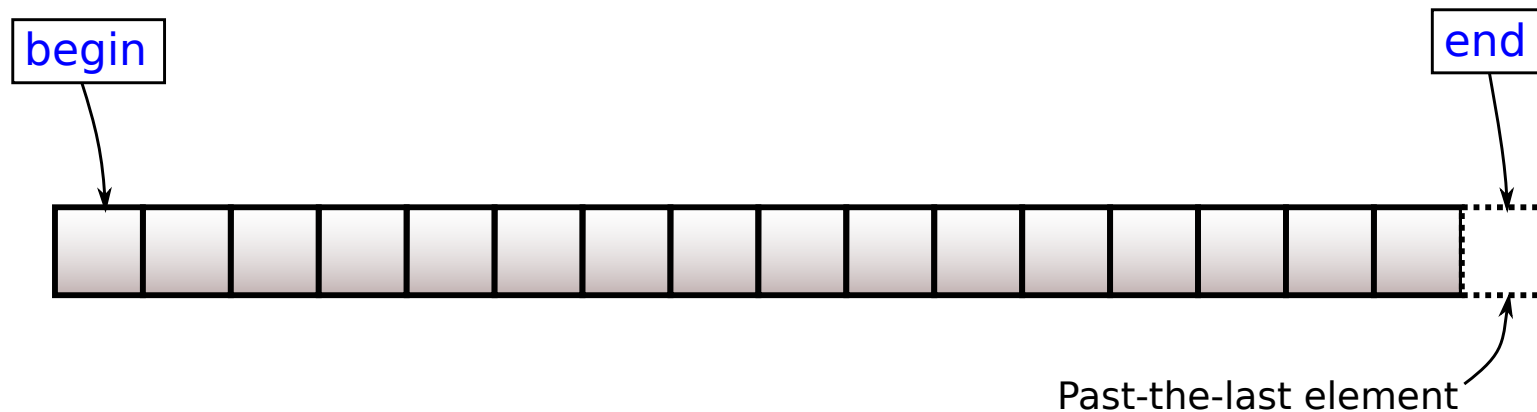
Every container has its iterator: `Container::iterator` . e.g.

`std::vector<int>::iterator` , `std::forward_list<std::string>::iterator`

- `auto` comes to our rescue!

`c.begin()` returns the iterator to the first element of `c` .

`c.end()` returns the iterator to **the element following the last element** of `c` .



# 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** of the range, i.e. the number of elements. There is no extra `+1` or `-1` .
- If `b == e` , the range is empty.

# 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 `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::list<int> &lst) {
    for (auto it = lst.begin(); it != lst.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 `vector` may cause the reallocation of storage. All iterators obtained previously are invalidated.
- Deleting an element in a `list` will invalidate the iterator referring to that element.

# 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 point to any element 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 `<`, `<=`, `>`, `>=`.

# Iterator categories

**ForwardIterator**: supports `*it`, `it->mem`, `++it`, `it++`, `it1 == it2`, `it1 != it2`

**BidirectionalIterator**: a ForwardIterator that can be moved in both directions

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

**RandomAccessIterator**: a BidirectionalIterator that can be moved to point to any element 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 `<`, `<=`, `>`, `>=`.

\* Which category is the built-in pointer in?

# Iterator categories

**ForwardIterator**: supports `*it`, `it->mem`, `++it`, `it++`, `it1 == it2`, `it1 != it2`

**BidirectionalIterator**: a ForwardIterator that can be moved in both directions

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

**RandomAccessIterator**: a BidirectionalIterator that can be moved to point to any element 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 `<`, `<=`, `>`, `>=`.

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

# Iterator categories

**ForwardIterators:** an iterator that can be moved forward.

- `forward_list<T>::iterator`

**BidirectionalIterator:** a ForwardIterator that can be moved in both directions

- `list<T>::iterator`

**RandomAccessIterator:** a BidirectionalIterator that can be moved to point to any element in constant time.

- `string::iterator`, `vector<T>::iterator`, `deque<T>::iterator`,  
`array<T,N>::iterator`

# Iterator categories

To know the category of an iterator of a container, consult its type alias member `iterator_category`.

```
using vec_iter = std::vector<int>::iterator;  
using category = vec_iter::iterator_category;
```

Put your mouse on `category`, and the IDE will tell you what it is.

It is one of the following tags: `std::forward_iterator_tag`,  
`std::bidirectional_iterator_tag`, `std::random_access_iterator_tag`.

Note: There are two other categories: `InputIterator` and `OutputIterator`. They may (or may not) be covered in later lectures.

# Constructors of containers

All sequence containers can be constructed in the following ways:

- Container `c(b, e)` , where `[b, e)` is an **iterator range**.
  - Copies elements from the iterator range `[b, e)` .
- Container `c(n, x)` , where `n` is a nonnegative integer and `x` is a value.
  - Initializes the container with `n` copies of `x` .
- Container `c(n)` , where `n` is a nonnegative integer.
  - Initializes the container with `n` elements. All elements are **value-initialized**.
  - This is not supported by `string` . (Why?)



# Constructors of containers

All sequence containers can be constructed in the following ways:

- Container `c(b, e)` , where `[b, e)` is an **iterator range**.
  - Copies elements from the iterator range `[b, e)` .
- Container `c(n, x)` , where `n` is a nonnegative integer and `x` is a value.
  - Initializes the container with `n` copies of `x` .
- Container `c(n)` , where `n` is a nonnegative integer.
  - Initializes the container with `n` elements. All elements are **value-initialized**.
  - This is not supported by `string` , because it is meaningless to have `n` value-initializes `char` s (all of them will be `'\0'` )!

# Algorithms and function objects

# 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":

```
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

```
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` :

```
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:

```
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`.

# 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 *RandomAccessIterators*, while `std::copy` allows any *InputIterators*.

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.

```
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`

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

```
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, 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?

```
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:

```
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.

```
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.

```
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:

```
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;  
// 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 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.

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.

# Associative containers

# Motivation: set

Represent a "set":

- Quick insertion, lookup and deletion of elements.
- Order does not matter.

Sequence containers do not suffice:

- Lookup of elements is  $O(n)$ .
- Quick insertion/deletion only happens at certain positions for some containers.
  - e.g. `vector` only supports quick insertion/deletion at the end.
- The order of elements is preserved, which is not important.

You will learn the appropriate data structures in CS101.

# std::set

Defined in `<set>`.

- `std::set<T>` is a set whose elements are of type `T`. `operator<(const T, const T)` **should be supported**, because it is usually implemented as Red-black trees.
- `std::set<T, Cmp>` is also available. `x < y` will be replaced with `cmp(x, y)`, where `cmp` is a function object of type `Cmp`.

```
std::set<int> s1; // An empty set of ints
std::set<std::string> s2{"hello", "world"}; // A set of strings,
                                           // initialized with two elements

struct Student { std::string name; int id; };
std::set<Student> s3; // No operator< for Student is available.
                    // This line alone does not cause error, but you cannot
                    // insert elements into it.
s3.insert(Student{"Alice", 42}); // Error: No operator< available.
```

## `std::set`

Defined in `<set>`.

- `std::set<T>` is a set whose elements are of type `T`. `operator<(const T, const T)` **should be supported**, because it is usually implemented as Red-black trees.
- `std::set<T, Cmp>` is also available. `x < y` will be replaced with `cmp(x, y)`, where `cmp` is a function object of type `Cmp`.

```
struct Student { std::string name; int id; };
struct CmpStudentByName {
    bool operator()(const Student &a, const Student &b) const {
        return a.name < b.name;
    }
};
std::set<Student, CmpStudentByName> students; // OK
students.insert(Student{"Alice", 42}); // OK
```



# `std::set`

## Constructors

```
std::set<Type> s1{a, b, c, ...};  
std::set<Type> s2(begin, end); // An iterator range [begin, end)
```

C++17 CTAD (Class Template Argument Deduction) also applies:

```
std::set s1{a, b, c, ...}; // Element type is deduced according to the list  
std::set s2(begin, end); // Element type is deduced according to  
                        // the type of elements pointed by `begin` and `end`.
```

Besides, `std::set` is copy-constructible, copy-assignable, move-constructible and move-assignable, just as the sequence containers we have learned.

`std::set` does not contain duplicate elements. These constructors will ignore duplicate elements.

## `std::set`: operations

Common operations: `s.empty()`, `s.size()`, `s.clear()`.

Insertion: `insert` and `emplace`. Duplicate elements will not be inserted.

- `s.insert(x)`, `s.insert({a, b, ...})`, `s.insert(begin, end)`.

```
std::set s{3, 2, 5, 5, 1}; // {1, 2, 3, 5}. The duplicate 5 is removed.
std::cout << s.size() << std::endl; // 4
s.insert(42); // {1, 2, 3, 5, 42}
s.insert(42); // Nothing is inserted. (No errors.)
int a[]{10, 20, 30};
s.insert(a, a + 3); // An iterator range.
                    // s now contains {1, 2, 3, 5, 10, 20, 30, 42}.
s.insert({11, 12}); // {1, 2, 3, 5, 10, 11, 12, 20, 30, 42}.
```

## `std::set`: insertion

Insertion: `insert` and `emplace`. Duplicate elements will not be inserted.

- `s.emplace(args...)`. Forwards the arguments `args...` to the constructor of the element type, and constructs the element in-place.

```
std::set<std::string> s;  
s.emplace(10, 'c'); // inserts a string "ccccccccc"
```

`s.insert(x)` and `s.emplace(args...)` returns `std::pair<iterator, bool>`:

- On success, `.first` is an `iterator` pointing to the inserted element, and `.second` is `true`.
- On failure, `.first` is an `iterator` pointing to the element that prevented the insertion, and `.second` is `false`.

## `std::set`: iterators

`s.begin()` , `s.end()` : Begin and off-the-end iterators.

The iterator of `std::set` is **BidirectionalIterator**:

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

**The elements are in ascending order:** The following assertion always succeeds (if both `tmp` and `iter` are dereferenceable).

```
auto tmp = iter;  
++iter;  
assert(*tmp < *iter);
```

## `std::set`: iterators

Elements in a `set` cannot be modified directly: `*iter` returns a reference-to-`const`.

- The elements are stored in specific positions in the red-black tree, according to their values.
- You cannot change their values arbitrarily.

## std::set: traversal

Range-for still works!

```
std::set<int> s{5, 5, 7, 3, 20, 12, 42};  
for (auto x : s)  
    std::cout << x << ' '  
std::cout << std::endl;
```

Output: 3, 5, 7, 12, 20, 42 . The elements are in ascending order.

Equivalent way: Use iterators

```
for (auto it = s.begin(); it != s.end(); ++it)  
    std::cout << *it << ' '  
std::cout << std::endl;
```

## std::set: deletion

Delete elements: `erase`

- `s.erase(x)` , `s.erase(pos)` , `s.erase(begin, end)` , where `pos` is an iterator pointing to some element in `s` , and `[begin, end)` is an iterator range in `s` .
- `s.erase(x)` removes the element that is equivalent to `x` , **if any**.
  - returns `0` or `1` , indicating the number of elements removed.

```
std::set<int> s{5, 5, 7, 3, 20, 12, 42};  
std::cout << s.erase(42) << std::endl; // 42 is removed. output: 1  
// s is now {3, 5, 7, 12, 20}.  
s.erase(++s.begin()); // 7 is removed.
```

## `std::set`: element lookup

`s.find(x)` , `s.count(x)` , and some other functions.

`s.find(x)` returns an iterator pointing to the element equivalent to `x` (if found), or `s.end()` (if not found).

```
std::set<int> s = someValues();  
if (s.find(x) != s.end()) // x is found  
    // ...
```



## `std::set`: pros and cons

The time complexity of insertion, deletion, and lookup of elements in a `std::set` : logarithmic in the size of the container. ( $O(\log n)$ )

- Compared to sequence containers, this is (almost) a huge improvement.

Elements are sorted automatically.

Fast random access like `v[i]` is not supported.

## Other kinds of sets:

Sets based on red-black trees:

- `std::set`
- `std::multiset` : allows duplicate elements

Sets based on hash-tables: (since C++11)

- `std::unordered_set` : hash-table version of `std::set`
- `std::unordered_multiset` : allows duplicate elements

Sets based on hash-tables provides (average-case)  $O(1)$  time operations, but requires the data to be hashable.

## Motivation: map

Represent a map:  $f : S \rightarrow T$ .

- For sequence containers `Container<Type>`:  $S = \{0, 1, 2, \dots, N - 1\}$  (index),  $T$  is the set of values of type `Type`.
- For `std::set<Type>`:  $T = \{\text{exist}, \text{not-exist}\}$ ,  $S$  is the set of values of type `Type`.

`std::map<Key, Value>`: defined in `<map>`

- `Key` is the type of elements in  $S$ , and `Value` is the type of elements in  $T$ .
- Stores "key-value" pairs.

## Motivation: map

Example: Count the occurrences of strings.

```
std::map<std::string, int> counter; // maps every string to an integer
std::string word;
while (std::cin >> word)
    ++counter[word]; // !!
```

Now for any string `str`, `counter[str]` is an integer indicating how many times `str` has occurred.

## `std::map`: comparison with `std::set`

`std::map<Key, Value>` has two template parameters: `Key` and `Value`.

- If we ignore `Value`, it is a `std::set<Key>`.
  - Duplicate keys are not allowed.
  - `operator<(const Key, const Key)` is required.
  - Elements are stored in **ascending order of keys**.
  - Keys cannot be modified directly.
- The element type of `std::map<Key, Value>` is `std::pair<const Key, Value>`.
  - `*iter` returns `std::pair<const Key, Value> &`.

# `std::map`: comparison with `std::set`

Constructors:

- `std::map<Key, Value> m{{key1, value1}, {key2, value2}, ...};`
- `std::map<Key, Value> m(begin, end)` , but the elements should be pairs:

```
std::vector<std::pair<int, int>> v{{1, 2}, {3, 4}};  
std::map<int, int> m(v.begin(), v.end());
```

Insertion:

- `m.insert({key, value})`
- `m.insert({{key1, value1}, {key2, value2}, ...})`
- `m.insert(begin, end)`

## `std::map`: comparison with `std::set`

Deletion:

- `m.erase(pos)`, `m.erase(begin, end)`: same as `std::set<T>::erase`.
- `m.erase(key)`: Removes the element whose *key* is `key`.

Iterators: **BidirectionalIterator**, pointing to `std::pair<const Key, Value>`.

```
std::map<std::string, int> counter = someValues();  
for (auto it = counter.begin(); it != counter.end(); ++it)  
    std::cout << it->first << " occurred " << it->second << " times.\n";
```

## `std::map`: traversal

Use range-for:

```
for (const auto &kvpair : counter)
    std::cout << kvpair.first << " occurred " << kvpair.second << " times.\n";
```

It's so annoying to deal with the `pair` stuff...



## `std::map`: traversal

Use range-for:

```
for (const auto &kvpair : counter)
    std::cout << kvpair.first << " occurred " << kvpair.second << " times.\n";
```

It's so annoying to deal with the `pair` stuff...

**C++17 structured binding** kills the game!

```
for (const auto &[word, occ] : counter)
    std::cout << word << " occurred " << occ << " times.\n";
```

(Looks very much like Python unpacking.)



## `std::map`: element lookup

`m.find(key)`, `m.count(key)`, and some other member functions.

Note: `m.find(key)` does not insert elements. `m[key]` will insert an element if that *key* does not exist.

## Other kinds of maps:

Maps based on red-black trees:

- `std::map`
- `std::multimap`: allows duplicate elements

Maps based on hash-tables: (since C++11)

- `std::unordered_map`: hash-table version of `std::map`
- `std::unordered_multimap`: allows duplicate elements

Maps based on hash-tables provides (average-case)  $O(1)$  time operations, but requires the *key* to be hashable.