CS100 Recitation 12

GKxx

May 16, 2022

Contents

Contents

1. Sequential Containers

2. Algorithms

Overview

Sequential containers from old STL:

`vector`	Flexible-size array.
`deque`	Double-ended queue.
`list`	Doubly-linked list.
`string`	Specialized container for strings.

Sequential containers added in C++11:

`forward_list`	Singly-linked list.
`array`	Encapsulates built-in array.

Note: `array` requires an additional template argument: `array<T, N>`, where `N` must be an integer value known at compile-time.

Type Aliases

- value_type
- size_type: Return-type of size().
- difference_type: Return-type of subtracting two iterators.
- pointer: value_type *.
- reference: value_type &.
- const_pointer: const value_type *.
- const_reference: const value_type &.
- iterator
- const_iterator: cannot modify the elements.

Obtaining Iterators

Notes:

- On a const container, begin() and end() return const_iterators.
- cbegin() and cend() were added into the C++ standard since C++11.

```
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;
```

Construction

`C c;`	Default construction.
`C c1(c2);`	Construct `c1` as a copy of `c2`.
`C c(b, e);`	Copy elements from the iterator range `[b,e)`.
`C c{a,b,c,d,...}`	List initialization.
`C c(n);`	`c` has `n` value-initialized elements.
`C c(n, x);`	`c` has `n` copies of `x`.

Notes:

- Default construction for `array`: Default-initialization of every element.
- Construction from an iterator range is not valid for `array`.
- `C c(n);` is not valid for `string` or `array`.
- `C c(n, x);` is not valid for `array`.

Construction

Copy elements from a `list` to initialize a `vector`?

Construction

Copy elements from a `list` to initialize a `vector`?

```
std::list<int> l = some_value();
std::vector<int> v(l.begin(), l.end());
```

Construction

Copy elements from a list to initialize a vector?

```
std::list<int> l = some_value();
std::vector<int> v(l.begin(), l.end());
```

What about copying them in reverse order?

Construction

Copy elements from a `list` to initialize a `vector`?

```
std::list<int> l = some_value();
std::vector<int> v(l.begin(), l.end());
```

What about copying them in reverse order?

```
std::vector<int> v2(l.rbegin(), l.rend());
```

- `rbegin()`, `rend()`, `crbegin()`, `crend()`
- `reverse_iterator`, `const_reverse_iterator`
- Not valid for `forward_iterator`.

Inserting and Erasing Elements

	push/pop_back	push/pop_front	insert/erase
vector	✓	✗	slow
deque	✓	✓	slow
list	✓	✓	✓
forward_list	✗	✓	✗
array	✗	✗	✗

`insert` and `erase`

Not for `array` or `forward_list`:

`c.insert(it, x)`	Insert `x` **before** `it`.
`c.erase(it)`	Erase the element at position `it`.

For `forward_list`:

`l.insert_after(it, x)`	Insert `x` **after** `it`.
`l.erase_after(it)`	Erase the element **after** the position `it`.

Notes:

- `it` can be `iterator` or `const_iterator`.
- There are many overloads:
    - `c.insert(it, n, x)`: Insert `n` copies of `x`.
    - `c.insert(it, b, e)`: Insert elements copied from iterator range.

Emplace

With *variadic templates*, *universal references* and *perfect forwarding*, C++11 introduces the 'emplace' operations:

```
struct Point2d {
  Point2d(double, double);
};
std::vector<Point2d> vp;
vp.emplace_back(3.5, 6);
std::deque<std::string> ds;
ds.emplace_front(10, 'c');
```

- `emplace`, `emplace_back`, `emplace_front`.
- `value_type` need not to be copy-constructible or copy-assignable.

Equality and Relational Operators

`==`, `!=`, `<`, `<=`, `>`, `>=`.

- `==` and `!=` only rely on `operator==` of `value_type`.
- `<`, `<=`, `>`, `>=` only rely on `operator<` of `value_type`.
- Minimize the requirements on unknown types!

Equality and Relational Operators

`==`, `!=`, `<`, `<=`, `>`, `>=`.

- `==` and `!=` only rely on `operator==` of `value_type`.
- `<`, `<=`, `>`, `>=` only rely on `operator<` of `value_type`.
- Minimize the requirements on unknown types!
- `std::equal` and `std::lexicographical_compare`.

Iterator Categories

Iterators are classified into **five categories**: input-iterator, output-iterator, forward-iterator, bidirectional-iterator, random-access-iterator.

- `vector`, `deque`, `string` and `array` have random-access-iterators.
- `list` has bidirectional-iterator.
- `forward_list` has forward-iterator.

Iterator Categories

Iterators are classified into **five categories**: input-iterator, output-iterator, forward-iterator, bidirectional-iterator, random-access-iterator.

- vector, deque, string and array have random-access-iterators.
- list has bidirectional-iterator.
- forward_list has forward-iterator.

Operations:

- A forward-iterator supports operator* (dereference), operator++ (prefix and postfix incrementation), operator== and operator!=.
- A bidirectional-iterator **is a** forward-iterator, and it also supports oprator-- (prefix and postfix decrementation).
- A random-access-iterator **is a** bidirectional-iterator, and it also supports it1-it2, it+n, it-n, n+it, +=, -=, it[n] and <, <=, >, >=.

Iterator Categories

Defined in <iterator>:

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag : input_iterator_tag {};
  struct bidirectional_iterator_tag : forward_iterator_tag {};
  struct random_access_iterator_tag
      : bidirectional_iterator_tag {};
}
```

Iterator Categories

Defined in `<iterator>`:

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag : input_iterator_tag {};
  struct bidirectional_iterator_tag : forward_iterator_tag {};
  struct random_access_iterator_tag
      : bidirectional_iterator_tag {};
}
```

Every STL iterator has a type alias member `iterator_category`, which is one of the five tags.

- e.g. `vector<int>::iterator::iterator_category` is `std::random_access_iterator_tag`.
- What are they used for?

Container Adapters

stack, queue and priority_queue are ‘container adapters’.

- They are NOT containers and have no iterators.
- They use a container to store data, and re-define the interfaces to resemble the corresponding data structures.

```
void bfs() {
  std::queue<int> q;
  q.push(s); vis[s] = true;
  while (!q.empty()) {
    int x = q.front(); q.pop();
    for (auto i = head[x]; i; i = next[i])
      if (!vis[v[i]]) {
        q.push(v[i]);
        vis[v[i]] = true;
      }
  }
}
```

vector<bool>

It's not necessarily *bad*, but you should be very careful when using it.

Possible substitutions:

- std::deque<bool>
- std::bitset
- boost::dynamic_bitset

Contents

1 Sequential Containers

2 Algorithms

Algorithms in STL

Sort a vector, drop duplicates, and obtain the number of different values.

```
std::vector<int> v = some_value();
std::sort(v.begin(), v.end());
auto it = std::unique(v.begin(), v.end());
int n = it - v.begin();
```

Algorithms in STL

See *C++ Primer* Appendix A.2.

- Iterator ranges
- Predicates.

Customize Operations

Sort the `vector<Point2d>` in order of the `x` coordinate.

Customize Operations

Sort the `vector<Point2d>` in order of the `x` coordinate.

- Overload `operator<` for `Point2d`?

Customize Operations

Sort the vector<Point2d> in order of the x coordinate.

- Overload operator< for Point2d?
- Pass a comparator function:

```
inline bool comp(const Point2d &lhs, const Point2d &rhs) {
  return lhs.get_x() < rhs.get_x();
}
std::sort(v.begin(), v.end(), comp);
```

Customize Operations

Sort the `vector<Point2d>` in order of the `x` coordinate.

- Overload `operator<` for `Point2d`?
- Pass a comparator function:

```
inline bool comp(const Point2d &lhs, const Point2d &rhs) {
  return lhs.get_x() < rhs.get_x();
}
std::sort(v.begin(), v.end(), comp);
```

It's better to write it as a `static` function of `Point2d`:

```
struct Point2d {
  static bool cmp_x(const Point2d &lhs, const Point2d &rhs) {
    return lhs.get_x() < rhs.get_x();
  }
};
std::sort(v.begin(), v.end(), Point2d::cmp_x);
```

Customize Operations

Find the first element less than 10:

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

Customize Operations

Find the first element less than 10:

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

Find the first element less than `k`? (`k` is runtime-determined)

Customize Operations

Find the first element less than 10:

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

Find the first element less than `k`? (`k` is runtime-determined)

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

Overloading operator()

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

- `Less_than(k)` creates an object of the type `Less_than`.
- `lt(x)` is equivalent to `lt.operator()(x)`, which returns true when `x < lt.k`.

Overloading operator()

Rewrite Point2d::cmp_x:

```
struct Cmp_x {
  bool operator()(const Point2d &lhs, const Point2d &rhs) const {
    return lhs.get_x() < rhs.get_x();
  }
};
std::sort(v.begin(), v.end(), Cmp_x{});
```

or equivalently,

```
std::sort(v.begin(), v.end(), Cmp_x());
```

Lambda: The First Glance

An anonymous function.

```
std::sort(v.begin(), v.end(),
    [](const Point2d &lhs, const Point2d &rhs) {
      return lhs.get_x() < rhs.get_x();
    });
```

Lambda: The First Glance

An anonymous function.

```
std::sort(v.begin(), v.end(),
    [](const Point2d &lhs, const Point2d &rhs) {
      return lhs.get_x() < rhs.get_x();
    });
```

Capture a variable:

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

Lambda: The First Glance

```
auto f = [k](int x) { return x < k; };
f(k)     // false
f(k - 1) // true
```

What's the type of a lambda?

Lambda: The First Glance

```
auto f = [k](int x) { return x < k; };
f(k)     // false
f(k - 1) // true
```

What's the type of a lambda?

No one but the compiler knows.

Callable in C++

Callable in C:

- functions
- pointers to functions

Callable in C++

Callable in C:

- functions
- pointers to functions

Callable in C++:

- functions
- pointers to functions
- lambdas
- objects that have an `operator()` member.