CS100 Recitation 13

GKxx

May 23, 2022

Contents

6 Components

- Containers
  - Sequential containers: `list`, `vector`, `deque`, ...
  - Associative containers
    - Ordered: `map`, `set`, `multimap`, `multiset` **(RB-tree)**
    - Unordered: `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset` **(Hash-table)**
- Iterators
- Algorithms
- Adapters
  - Container adapters: `stack`, `queue`, `priority_queue`
  - Iterator adapters: Insert iterators, stream iterators, reverse iterators, move iterators
- Allocators
- Functors
  - `std::plus`, `std::minus`, `std::less`, ...
  - `std::function`, `std::bind`, ...

Contents

Sets

副本

Maintain a set of integers. Duplicates should be dropped.

```
std::vector<int> v;
int x;
while (std::cin >> x) {
  bool found = false;
  for (auto y : v)
    if (y == x) {
      found = true;
      break;
    }
  if (!found)
    v.push_back(x);
}
```

Sets

Use `std::find`:

```
while (std::cin >> x) {
  auto found = std::find(v.begin(), v.end(), x);
  if (found == v.end())
    v.push_back(x);
}
```

Still inefficient...

Sets

Use `std::find`:

```
while (std::cin >> x) {
  auto found = std::find(v.begin(), v.end(), x);
  if (found == v.end())
    v.push_back(x);
}
```

Still inefficient...

```
std::set<int> s;
int x;
while (std::cin >> x)
  s.insert(x);
```

Maps

Map student numbers to scores:

```
int scores[N];
scores[3] = 90;
```

Map student names to scores?

Maps

Map student numbers to scores:

```
int scores[N];
scores[3] = 90;
```

Map student names to scores?

```
std::map<std::string, int> scores;
scores["Alice"] = 90;
```

Contents

Insert Iterators

Copy elements from a `vector` to another:

```
std::copy(v.begin(), v.end(), v2.begin());
```

But this requires `v2` to have enough room.

Insert Iterators

Copy elements from a `vector` to another:

```
std::copy(v.begin(), v.end(), v2.begin());
```

But this requires `v2` to have enough room.

```
v2.resize(v.size());
std::copy(v.begin(), v.end(), v2.begin());
```

Insert Iterators

Copy elements from a `vector` to another:

```
std::copy(v.begin(), v.end(), v2.begin());
```

But this requires `v2` to have enough room.

```
v2.resize(v.size());
std::copy(v.begin(), v.end(), v2.begin());
```

Copy elmeents **to the end** of another `vector`?

Insert Iterators

Copy elements from a `vector` to another:

```
std::copy(v.begin(), v.end(), v2.begin());
```

But this requires `v2` to have enough room.

```
v2.resize(v.size());
std::copy(v.begin(), v.end(), v2.begin());
```

Copy elmeents **to the end** of another `vector`?

```
std::copy(v.begin(), v.end(), std::back_inserter(v2));
```

Stream Iterators

Write elements to the output stream?

```
std::ostream_iterator<int> out_iter(std::cout, " ");
std::copy(v.begin(), v.end(), out_iter);
```

Stream Iterators

Write elements to the output stream?

```
std::ostream_iterator<int> out_iter(std::cout, " ");
std::copy(v.begin(), v.end(), out_iter);
```

Construct a `vector` from input:

```
std::istream_iterator<int> in_iter(std::cin), eof;
std::vector<int> vec(in_iter, eof);
```

Use `back_inserter` together:

```
std::copy(in_iter, eof, std::back_inserter(vec));
```

Stream Iterators

- `std::istream_iterator` is of the category input-iterator.
- `std::ostream_iterator` is of the category output-iterator.

Difference between forward-iterators and input-iterators?

Stream Iterators

- `std::istream_iterator` is of the category input-iterator.
- `std::ostream_iterator` is of the category output-iterator.

Difference between forward-iterators and input-iterators?
Input-iterators are disposable, while forward-iterators provide *multi-pass guarantee*.

```
auto tmp = iter;
auto value = *iter;
++tmp;
// *iter == value ?
```

Contents

Sort

Using `std::sort` on `std::list` is not allowed...

- `std::sort` requires random-access-iterators, while `std::list` provides bidirectional-iterators.
- `std::sort` uses **Intro-sort**, which is a combination of **Quick-sort**, **Heap-sort** and **Insertion-sort**.

Sort

Using `std::sort` on `std::list` is not allowed...

- `std::sort` requires random-access-iterators, while `std::list` provides bidirectional-iterators.
- `std::sort` uses **Intro-sort**, which is a combination of **Quick-sort**, **Heap-sort** and **Insertion-sort**.

Use `std::list::sort` instead!

```
std::list<int> l = some_value();
l.sort();
```

- Value type is not required to be swappable.
- Stable sort.

Special Operations for Linked-lists

See *C++ Primer* Chapter 10.6.

Table 10.6. Algorithms That are Members of list and forward_list

	These operations return `void`.
`lst.merge(lst2)` `lst.merge(lst2, comp)`	Merges elements from `lst2` onto `lst`. Both `lst` and `lst2` must be sorted. Elements are removed from `lst2`. After the `merge`, `lst2` is empty. The first version uses the `<` operator; the second version uses the given comparison operation.
`lst.remove(val)` `lst.remove_if(pred)`	Calls `erase` to remove each element that is `==` to the given value or for which the given unary predicate succeeds.
`lst.reverse()`	Reverses the order of the elements in `lst`.
`lst.sort()` `lst.sort(comp)`	Sorts the elements of `lst` using `<` or the given comparison operation.
`lst.unique()` `lst.unique(pred)`	Calls `erase` to remove consecutive copies of the same value. The first version uses `==`; the second uses the given binary predicate.

Contents

Allocators

Consider implementing a `vector`...

- Separate memory allocation/deallocation and object construction/destruction.
- Adopt possibly different methods of memory allocation?

Allocators

Consider implementing a `vector`...

- Separate memory allocation/deallocation and object construction/destruction.
- Adopt possibly different methods of memory allocation?

Use different kinds of allocators that provide the same interfaces.

std::allocator

See *C++ Primer* Chapter 12.2.2.

Table 12.7. Standard allocator Class and Customized Algorithms

`allocator<T> a`	Defines an `allocator` object named `a` that can allocate memory for objects of type `T`.
`a.allocate(n)`	Allocates raw, unconstructed memory to hold `n` objects of type `T`.
`a.deallocate(p, n)`	Deallocates memory that held `n` objects of type `T` starting at the address in the `T*` pointer `p`; `p` must be a pointer previously returned by `allocate`, and `n` must be the size requested when `p` was created. The user must run `destroy` on any objects that were constructed in this memory before calling `deallocate`.
`a.construct(p, args)` deprecated since C++17 removed since C++20	`p` must be a pointer to type `T` that points to raw memory; *args* are passed to a constructor for type `T`, which is used to construct an object in the memory pointed to by `p`.
`a.destroy(p)` deprecated since C++17 removed since C++20	Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the `T*` pointer `p`.

Allocators

`std::vector<T>` uses `std::allocator<T>` by default.

```
std::vector<int> v1;
std::vector<int, std::allocator<int>> v2;
std::vector<int, gkxx::Simple_allocator<int>> v3;
```

Contents

Function-Call Operator

```
inline int adder(int a, int b) {
  return a + b;
}
int x = adder(42, 35);
```

The expression '`adder(42, 35)`' consists of one operator and **three** operands:

- `()` is the **function-call** operator.
- Three operands are `adder`, `42` and `35`.

运算对象

Function Objects (Functors)

```
struct Adder {
  int operator()(int a, int b) const {
    return a + b;
  }
};
int x = Adder{}(42, 35);
int y = Adder()(42, 35); // Equivalent
```

Default-construct an `Adder` object first, and then call `operator()`.

- **Function objects** or **Functors**: Objects of classes that define the call operator.
- "Act like functions".

Functors

```
struct Add_k {
  int k;
  Add_k(int x) : k(x) {}
  void operator()(int &x) const {
    x += k;
  }
};
int x = 42;
Add_k{5}(x); // x becomes 47.
std::vector<int> v = some_value();
std::for_each(v.begin(), v.end(), Add_k{10});
```

Contents

Arithmetic and Relational Operators

```
class Rational {
 public:
  Rational(int x) : m_num(x), m_denom(1) {}
  Rational() : Rational(0) {}
  double to_double() const {
    return (double)m_num / m_denom;
  }
  // other members
};
inline bool operator==
    (const Rational &lhs, const Rational &rhs) {
  return
    std::fabs(lhs.to_double()-rhs.to_double()) < 1e-9;
}
```

Arithmetic and Relational Operators

```
inline bool operator!=
    (const Rational &lhs, const Rational &rhs) {
  return !(lhs == rhs); // Let operator== do the work.
}
```

Arithmetic and Relational Operators

```
inline bool operator!=
    (const Rational &lhs, const Rational &rhs) {
  return !(lhs == rhs); // Let operator== do the work.
}
```

Do we define them as members or non-members?

Arithmetic and Relational Operators

```
inline bool operator!=
    (const Rational &lhs, const Rational &rhs) {
  return !(lhs == rhs); // Let operator== do the work.
}
```

Do we define them as members or non-members?

```
Rational r = some_value();
if (r == 0)
  // do something.
```

Member or Non-member?

```
class Rational {
 public:
  bool operator==(const Rational &rhs) const {
    // ...
  }
  // other members
};
Rational r = some_value();
if (0 == r) // ERROR!
  // do something.
```

Member or Non-member?

```
class Rational {
 public:
  bool operator==(const Rational &rhs) const {
    // ...
  }
  // other members
};
Rational r = some_value();
if (0 == r) // ERROR!
  // do something.
```

Effective C++ Item 24: Declare non-member functions when type conversions should apply to all parameters.

Contents

Prefix and Postfix

Increment/decrement operators are preferred to be members, though not required by the standard.

```
class Rational {
 public:
  Rational &operator++() {
    m_num += m_denom;
    return *this;
  }
  Rational operator++(int) {
    auto tmp = *this;
    ++*this;
    return tmp;
  }
};
```

- Prefix `operator++` returns reference to `*this`,
- while postfix `operator++` returns copy of the object before incrementation.
- Postfix `operator++` has an extra parameter of type `int`, which is just a tag.

Contents

Dereference and Arrow Operators

```
template <typename T, bool is_const>
class Slist_iterator {
 public:
  reference operator*() const {
    // return reference to the data
    // denoted by this iterator
  }
  pointer operator->() const {
    return std::addressof(operator*());
  }
};
```

- To make `(*p).mem` equivalent to `p->mem`, `operator->` is always defined as this.
- Why do we use `std::addressof`? (`<memory>`)

Contents

Type Conversion Operator

```
class Rational {
 public:
  operator double() const {
    return (double)m_num / m_denom;
  }
};
Rational r = some_value();
double x = r;
```

Conversion to bool

Recall that:

```
std::cin >> x;
if (std::cin)
  // input succeeded.
```

A conversion to bool?

```
class istream {
 public:
  operator bool() const {
    // ...
  }
};
```

Surprise!

```
class istream {
 public:
  operator bool() const {
    // ...
  }
};
```

With this conversion operator, the following code **compiles** happily!

```
int i;
std::cin << i; // ???
```

`explicit` Conversion

```
class istream {
 public:
  explicit operator bool() const {
    // ...
  }
};
```

Only allow explicit conversion to happen through this function.

- Using an expression as the operand of `operator&&`, `operator||` or `operator!`,
- or in the condition part of `operator?:`,
- or placing it in the condition part of `if`, `for`, `while`, `do-while` statements, **are also seen as explicit conversion to** `bool`.

Be Careful with Ambiguity

```
struct B;
struct A {
  A(const B &);
};
struct B {
  operator A() const;
};
B b;
A a(b); // ERROR! Which conversion?
```

- Never define both a constructor `A::A(const B &)` and a conversion `B::operator A() const`.
- Be careful with conversion to built-in types.

Call Overloaded Operator Functions Directly

- `a + b` may be equivalent to `operator+(a, b)` or `a.operator+(b)`.
- `++a` is equivalent to `a.operator++()`,
- while `a++` should be `a.operator++(0)`. (Pass any integer you like.)

Remember to differentiate between members and non-members.