

CS100 Recitation 13

GKxx

May 23, 2022

Contents

More on STL

- Associative Containers
- Iterator Adapters
- Special Algorithms
- Allocators

Operator Overloading

- Function-Call Operator
- Arithmetic and Relational Operators
- Increment and Decrement Operators
- Dereference and Arrow Operators
- Type Conversions

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

Sets

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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 elements **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 elements **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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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

Contents

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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

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

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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.

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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.

Compound Assignment Operators

```
class Rational {  
    public:  
        Rational &operator+=(const Rational &);  
        Rational &operator-=(const Rational &);  
        friend Rational operator-(const Rational &);  
};  
  
inline Rational operator-(const Rational &r) {  
    return Rational(-r.m_num, r.m_denom);  
}  
  
inline Rational &Rational::operator+=  
    (const Rational &other) {  
    // add other to *this  
    return *this;  
}
```

Compound Assignment Operators

Define other operators in terms of `operator+=` and `operator-` (negation).

```
inline Rational &Rational::operator-=  
    (const Rational &other) {  
    return *this += -other;  
}  
  
inline Rational operator+  
    (const Rational &lhs, const Rational &rhs) {  
    return Rational(lhs) += rhs;  
}  
  
inline Rational operator-  
    (const Rational &lhs, const Rational &rhs) {  
    return Rational(lhs) -= rhs;  
}
```


Contents

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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

More on STL

Associative Containers

Iterator Adapters

Special Algorithms

Allocators

Operator Overloading

Function-Call Operator

Arithmetic and Relational Operators

Increment and Decrement Operators

Dereference and Arrow Operators

Type Conversions

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.