TDDD38/726G82 - Advanced programming in C++

STL III

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 More on Iterators
2 Algorithms
3 More on Function Objects

1 More on Iterators
2 Algorithms
3 More on Function Objects

More on Iterators

Iterating arrays

```
int array[3] {1, 2, 3};
for (int e : array)
{
  cout << e << endl;
}
```

More on Iterators

Iterating arrays

- arrays don’t have member functions
- more specifically; they don’t have `begin` and `end` functions
- yet we can use range-based for-loops to iterate them

More on Iterators

Range-based for-loops

```
for (auto&& e : array)
{

  // ...

}
```

```
auto it {std::begin(array)};
auto last {std::end(array)};
for (; it != last; ++it)
{
  auto&& e{*it};
  // ...
}
```

More on Iterators

`std::begin` and `std::end`

- `std::begin(v)`
  - if `v` has a member `begin()`, return `v.begin()`
  - if `v` is an array, return a pointer to the first element
- `std::end(v)`
  - if `v` has a member `end()`, return `v.end()`
  - if `v` is an array, return a pointer to the element past the last

More on Iterators

Declarations

```
template <typename C>
auto begin(C& c) -> decltype(c.begin())
{ return c.begin(); }

template <typename T, size_t N>
T* begin(T (&a)[N])
{ return a; }
```

More on Iterators

Declarations

```
template <typename C>
auto end(C& c) -> decltype(c.end())
{ return c.end(); }

template <typename T, size_t N>
T* end(T (&a)[N])
{ return a + N; }
```

More on Iterators

Iterator utility functions

- `std::advance`
- `std::distance`
- `std::next`
- `std::prev`

More on Iterators

Utility

- These functions are more general (work with all iterators)
- beware; these utility functions comes with some extra cost, so think carefully before using them

More on Iterators

Reverse iterators

```
vector<int> v {1, 2, 3};
auto it {v.rbegin()};
auto end {v.rend()};

// will write "3 2 1 " to the terminal
for (; it != end; ++it)
{
  cout << *it << " ";
}
```

More on Iterators

Reverse iterators

- A special type of iterator that traverses the container backwards
- `rbegin()` will point to the last element in the container (so `end() != rbegin()`)
- `rend()` points to the element *before* the first one (so `begin() != rend()`)
- is only available for containers with *BidirectionalIterators*

More on Iterators

Const iterators

```
set<int> const s {3, 6, 9};
auto it {s.cbegin()};
auto end {s.cend()};
for (; it != end; ++it)
{
  // reading is ok!
  cout << *it << " ";
  // writing is not ok...
  *it = 5;
}
```

More on Iterators

Const iterators

- `const_iterator` is an iterator that can be used for only reading data
- disallows modification of the underlying object
- is the only way to use iterators on containers marked `const`

More on Iterators

Access functions

- `std::rbegin()` and `std::rend()`
- `std::cbegin()` and `std::cend()`
- You can even combine them:
- `std::crbegin()` and `std::crend()`

More on Iterators

Input stream iterators

```
std::istream_iterator<int> iit {std::cin};
std::istream_iterator<int> end {};
while (iit != end)
{
  cout << *iit++ << endl;
}
```

More on Iterators

Input stream iterators

- `std::istream_iterator` are iterators that read data from an input stream
- the template parameter defines what type should be read from the stream
- data is read from the stream when the iterator is incremented
- dereferencing returns a copy of last read object
- is an *InputIterator*

More on Iterators

Output stream iterators

```
std::ostream_iterator<int> oit {std::cout};
*oit++ = 5; // will write 5 to the terminal
```

More on Iterators

Output stream iterators

```
std::ostream_iterator<int> oit {std::cout, " "};

// will write "0 1 2 3 4 " to the terminal
for (int i{0}; i < 5; ++i)
{
  *out++ = i;
}
```

More on Iterators

Output stream iterators

- `std::ostream_iterator` are iterators that *write* data to some output stream
- template parameter defines what type is to be written
- data is written to the stream during assignment
- is an *OutputIterator*
- constructor takes an extra parameter that defines a delimiter string to insert after each write

Algorithms

Core of STL

- Everything in STL is based around algorithms and containers
- There are 110+ algorithms defined in the STL (exact amount depends on version)
- All algorithms operate on *iterator ranges*
- Uses lambdas and function objects heavily
- Defined (mostly) in `<algorithm>`

Algorithms

Algorithm categories

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Sorting operations
- Sorted range operations
- Set operations

Algorithms

Algorithm categories

- Heap operations
- Minmax operations
- Comparison operations
- Permutation operations
- Numeric operations (`<numeric>`)
- Uninitialized operations (`<memory>`)

Algorithms

`for_each`; the dull one of the bunch!

```
void put(int n)
{
  cout << n << endl;
}

int main()
{
  std::vector<int> v { 1, 2, 3 };
  std::for_each(std::begin(v), std::end(v), put);
}
```

Algorithms

Possible implementation

```
template <typename It, typename Function>
Function for_each(It first, It last, Function f)
{
  while (first != last)
  {
    f(*first++);
  }
  return f;
}
```

Algorithms

What will be printed?

```
std::vector<int> v {1, 2, 3};
auto f {[x = 0](int n) mutable
        {
          x += n;
          return x;
        }};
std::for_each(std::begin(v), std::end(v), f);
cout << f(0) << endl;
```

Algorithms

Answer

- `0` will be printed
- `x` is an internal variable accessible inside the lambda which will retain its value through each successive call to `f`
- however; `std::for_each` takes `f` as a copy, so `f` will not have been called when we reach the print statement

Algorithms

Possible fix

```
std::vector<int> v {1, 2, 3};
auto f {[x = 0](int n) mutable
        {
          x += n;
          return x;
        }};
std::for_each(std::begin(v), std::end(v),
              std::ref(f));
cout << f(0) << endl;
```

Algorithms

`std::ref`

- `std::ref(x)` forces the compiler to interpret `x` as an lvalue
- thus forcing any template-parameters to deduce it as a reference parameter rather than a by-value parameter

Algorithms

Another possible fix

```
std::vector<int> v {1, 2, 3};
auto f {[x = 0](int n) mutable
        {
          x += n;
          return x;
        }};
f = std::for_each(std::begin(v), std::end(v), f);
cout << f(0) << endl;
```

Algorithms

`std::find`

```
std::vector<int> v {5, -2, 8, 4, 7};
auto it {
  std::find(std::begin(v), std::end(v), 8)
};
if (it == std::end(v))
{
  // we didn't find it :(
}
else
{
  cout << *it << endl;
}
```

Algorithms

Predicate algorithms

```
std::set<int> m { -1, 4, 0, 3 };
auto p {[](auto a)
        {
          return a >= 0;
        }};
if (std::all_of(std::begin(m), std::end(m), p))
{
  // all of the numbers are positive
}
```

Algorithms

Predicate algorithms

```
template <typename It,
          typename Predicate>
bool all_of(It first, It last, Predicate p)
{
  while (first != last)
  {
    if (!p(*first++))
    {
      return false;
    }
  }
  return true;
}
```

Algorithms

`std::copy`

```
std::vector<int> v {1, 2, 3};
std::set<int> s {};
std::copy(std::begin(v), std::end(v),
          std::inserter(s, std::end(s)));
```

Algorithms

A cool usage of `std::copy` and stream iterators

```
std::vector<int> v{1, 2, 3};
std::copy(std::begin(v), std::end(v),
          std::ostream_iterator<int>{cout, " "});
```

Algorithms

Another cool usage of `std::copy`

```
std::vector<int> v;
auto begin{std::istream_iterator<int>{cin}};
auto end{std::istream_iterator<int>{}};
std::copy(begin, end, std::back_inserter(v));
```

Algorithms

`std::transform`

```
std::vector<std::string> v{"1", "2", "3"};
std::vector<int> target{};

std::transform(std::begin(v), std::end(v),
               std::back_inserter(target),
               [](std::string const& s)
               {
                 return std::stoi(s);
               });
```

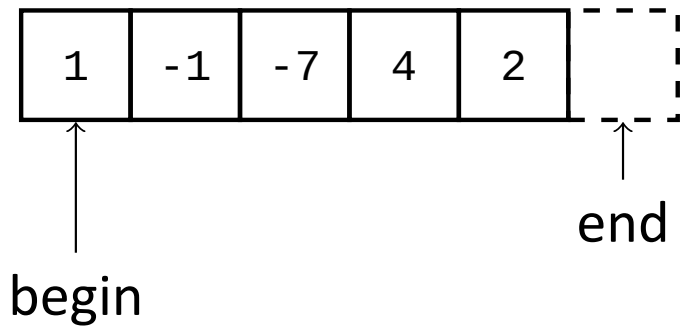
Algorithms

`std::remove_if`

```
std::vector<int> v{1, -1, -7, 4, 2};
v.erase(
    std::remove_if(std::begin(v), std::end(v),
                   [](int n)
                   {
                     return n < 0;
                   }), std::end(v));
```

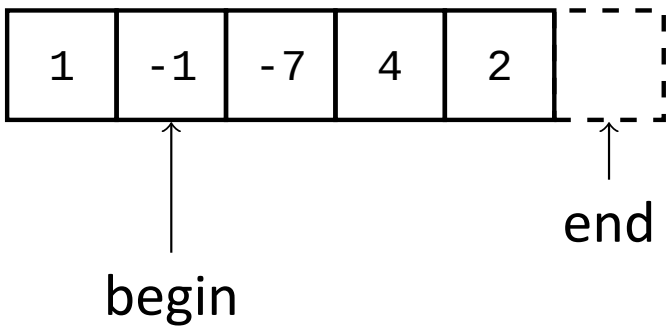
Algorithms

`std::remove_if`



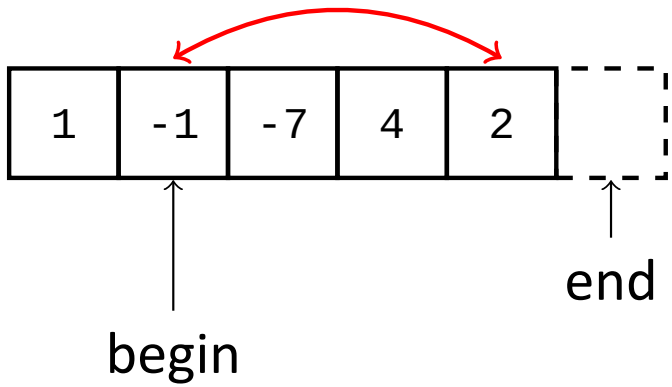
Algorithms

`std::remove_if`



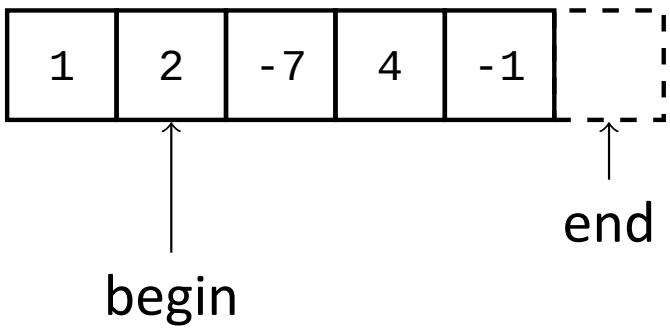
Algorithms

`std::remove_if`



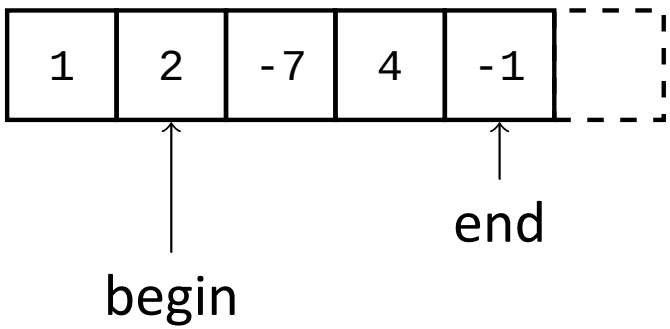
Algorithms

`std::remove_if`



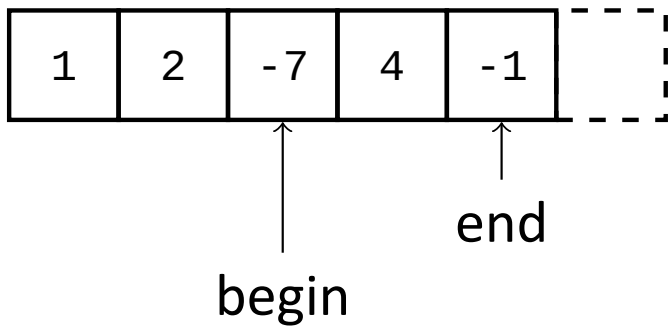
Algorithms

`std::remove_if`



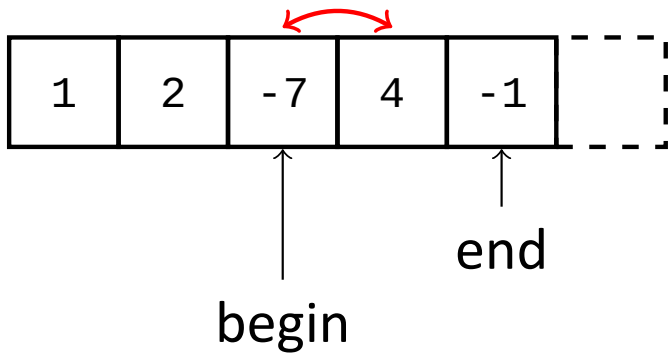
Algorithms

`std::remove_if`



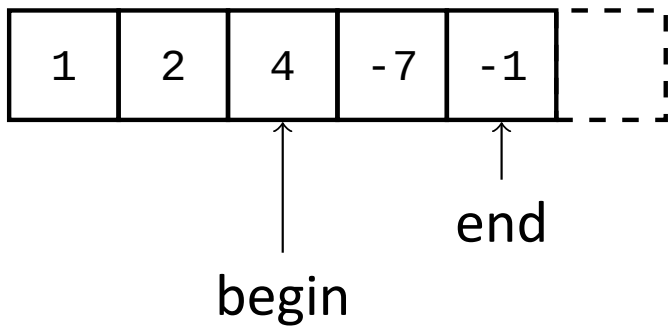
Algorithms

`std::remove_if`



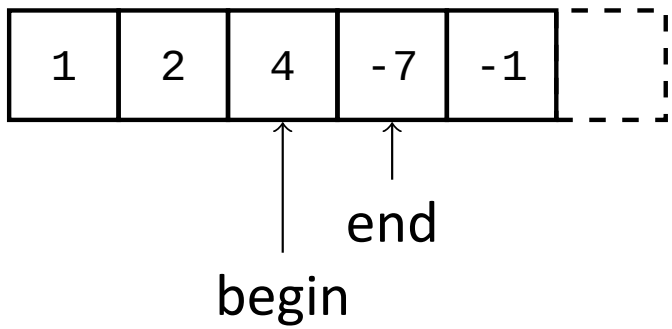
Algorithms

`std::remove_if`



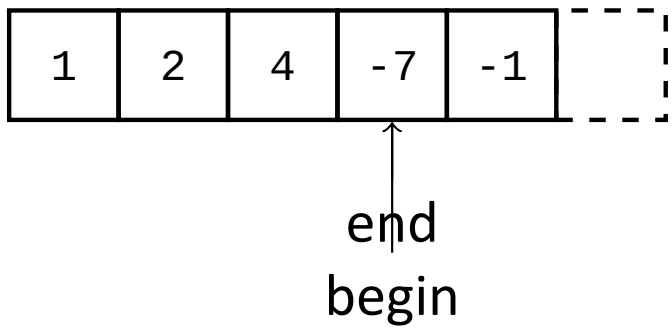
Algorithms

`std::remove_if`



Algorithms

`std::remove_if`



Algorithms

A note on modifying sequence operations

- algorithms operate on iterator ranges
- it is not possible to remove or add elements through arbitrary iterators
- due to this, no algorithm is able to remove elements from containers
- (they are able to add elements with the use of output iterators)

Algorithms

Modifying sequence operations

- All algorithms that are meant to remove elements will instead move them to the end of the range and return an iterator to the first removed element
- with that iterator we can now call the `erase` function of the underlying container to actually remove those elements

Algorithms

`std::accumulate`

```
#include <numeric>
// ...
int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};
  int sum{
    std::accumulate(std::begin(v), std::end(v), 0)
  };
  cout << sum << endl; // will print 15
}
```

Algorithms

`std::accumulate`

```
std::set<std::string> v{"1", "2", "3"};
int result{
  std::accumulate(std::begin(v), std::end(v), 4,
                  [](int n, std::string const& s)
                  {
                    return n + std::stoi(s);
                  })
};
```

Algorithms

`std::accumulate`

- `std::accumulate` is like a fold-expression
- but it operates during runtime on iterator ranges
- very flexible when combining values into a single value

Algorithms

Final words

- There are a lot of algorithms
- You are not expected to memorize them all
- However you must be able to find suitable algorithms and use them

Algorithms

Now go forth and use this great power!

LINKÖPING UNIVERSITY

More on Function Objects

std::function

```
#include <functional>
void foo() { }
struct Functor
{
  void operator()() { }
};
int main()
{
  std::function<void()> fun;
  fun = foo;
  fun = Functor{};
  fun = []() { };
}
```

More on Function Objects

`std::function`

- A type to represent any callable object
- Specify in the template parameter what signature the callable object must have
- can be used to store lambdas as variables

More on Function Objects

Some problems with `std::function`

```
void foo(int) { }
void bar(int, int = 0) { }
int main()
{
  std::function<void(int)> fun {foo}; // ok!
  fun = bar; // not ok

}
```

More on Function Objects

Some problems with `std::function`

- the specified signature must be an exact match
- it is not enough that the function we are trying to store is callable in the specified way

More on Function Objects

A possible solution

```
void foo(int) { }
void bar(int, int = 0) { }
int main()
{
  std::function<void(int)> fun {foo};
  fun = std::bind(bar, placeholders::_1, 0);
}
```

More on Function Objects

`std::bind`

- `std::bind` generates a function-object
- the function-object will be based on some other function-object
- with `std::bind` we can create lambdas that call functions where some or all of the arguments have specified values

More on Function Objects

`std::bind`

```
int add(int x, int y)
{
  return x + y;
}

int main()
{
  std::bind(add, 5, 10)(); // will return 15
}
```

More on Function Objects

`std::bind`

```
int main()
{
  auto f{std::bind(add, placeholders::_1, 10)};
  f(0); // will return 10
  f(10); // will return 20
}
```

More on Function Objects

`std::bind`

```
int sub(int a, int b)
{
  return a - b;
}
int main()
{
  using namespace std::placeholders;
  auto f{std::bind(sub, _2,_1)};
  f(10, 5); // will return -5
  f(2, 10); // will return 8
}
```

More on Function Objects

`std::bind`

- `placeholders` are used to specify which arguments should be *free*
- that is, which arguments should be available in the generated function
- `placeholders::_1` represents the first argument to the generated function
- `placeholders::_2` represents the second argument, and so on, up to some implemenation defined number

More on Function Objects

Another problem with `std::function`

```
struct Cls
{
  void foo() { }
};

int main()
{
  std::function<void()> fun;
  Cls c{};
  fun = c.foo; // not ok
  fun = &Cls::foo; // not ok
}
```

More on Function Objects

Why?

- Member functions are not ordinary functions
- Each member function requires an object to be called from
- Due to this, they cannot be bound to `std::function`

More on Function Objects

Solution

```
Cls c{};
auto foo{std::mem_fn(&Cls::foo)};
std::function<void()> fun;
fun = std::bind(foo, c);
```

More on Function Objects

`std::mem_fn`

- `std::mem_fn` converts a member function pointer to a normal function
- the generated function takes the object as a parameter
- therefore we can bind the result of `std::mem_fn` with the `Cls` object `c`
- this will make `fun()` equivalent to `c.foo()`

More on Function Objects

What will be printed?

```
int fun1()      { return 1; }
int fun2(int a) { return a * 3; }

int main()
{
  function<int()> fun{fun1};

  fun = bind([](int a, int b)
             {
                 return a + b;
             }, 1, fun());

  fun = [fun]() { return 2 * fun(); };

  cout << bind([](int x, int y, int z)
             {
                 return fun2(x) + y;
             }, _2, _1, 17)(fun(), 3);
}
```

www.liu.se

LINKÖPING UNIVERSITY