TDDD38/726G82 - Advanced programming in C++

STL II

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Iterators

General iterations

```
for (auto&& element : c)
{
  // ...
}
```

Iterators

General iterations

- iterating over a sequential container means going through all the elements in order
- semantically this is the same for all containers covered so far
- however; implementation varies wildly

Iterators

General iterations

- with `std::vector` and `std::array` we can iterate over the actual memory
- for `std::list` and `std::forward_list` we have to follow pointers
- `std::deque` is a combination of the two

Iterators

General iterations

- Writing general code using these containers requires abstraction
- we want one universal way of iterating over any container
- this has been solved with *iterators*

Iterators

General iterations

```
for (auto it{c.begin()}; it != c.end(); ++it)
{
  auto&& element{*it};
  // ...
}
```

Iterators

Iterators

```
auto it{c.begin()};
auto end{c.end()};
while (it != end)
{
  // ...
  ++it;
}
```

```
auto it{c.data()};
auto end{it + c.size()};
while (it != end)
{
  // ...
  ++it;
}
```

Iterators

Iterators

- iterators can be thought of as generalized pointers
- share similar interface and semantics with pointers
- However; no assumptions about the memory layout of elements
- the end-iterator signifies that the iteration is complete, i.e. we have visited all elements
- the end-iterator does *not* point to the last element, but rather one element past the last element

Iterators

Iterator categories

- *ForwardIterator*
  - can only step forward in the container
- *BidirectionalIterator*
  - can step forward and backwards in the container
- *RandomAccessIterator*
  - can access any element in the container

Iterators

InputIterator

```
std::vector<int> v{};
auto it{v.begin()};
for (int i{0}; i < 10; ++i)
{
  *it++ = i;
}
```

Iterators

InputIterator

```
std::vector<int> v{};
auto it{v.begin()};
for (int i{0}; i < 10; ++i)
{
  *it++ = i;
}
```

- Won’t work; `v.begin()` returns an *InputIterator*
- *InputIterators* can only access existing elements
- `v.insert` or `v.push_back` to add elements

Iterators

OutputIterator

- The standard library is built on using iterators
- Iterators define some kind of behaviour for various components
- Sometimes we want the iterators to add elements rather than modify existing ones
- This is where *OutputIterator*s come in

Iterators

OutputIterator

- *InputIterator*
  - \+ Can access elements in container
  - – Cannot add elements to container

Iterators

OutputIterator

- *InputIterator*
  - \+ Can access elements in container
  - – Cannot add elements to container
- *OutputIterator*
  - \+ Can add elements to container
  - – Cannot access elements in container

Iterators

OutputIterators

- *OutputIterator* is an *Iterator* so it must define `operator++` and `operator*`
- An *OutputIterator* cannot access elements so dereferencing the iterator shouldn't do anything
- Likewise shouldn't incrementing the iterator do anything
- The only operation that performs any work is `operator=` which will insert the right-hand side into the container

Iterators

`std::insert_iterator`

```
std::vector<int> v{};
auto it{std::inserter(v, v.end())};
for (int i{0}; i < 10; ++i)
{
  *it++ = i;
}
```

Iterators

`std::insert_iterator`

```
std::vector<int> v{};
auto it{std::inserter(v, v.end())};
for (int i{0}; i < 10; ++i)
{
  *it = i;
}
```

Iterators

`std::insert_iterator`

```
std::vector<int> v{};
auto it{std::inserter(v, v.end())};
for (int i{0}; i < 10; ++i)
{
  it = i;
}
```

Iterators

`std::insert_iterator`

```
std::vector<int> v{};
auto it{std::inserter(v, v.end())};
for (int i{0}; i < 10; ++i)
{
  v.insert(v.end(), i);
}
```

Iterators

`std::insert_iterator`

- Each assignment calls `insert` on the underlying container
- Is created with `std::inserter`
- must know which container it should operate on
- must know where in the container the insertion should happen
- Works with any container that has an `insert` function

Iterators

Other output iterators

- `std::back_inserter`
- `std::front_inserter`

Iterators

Other output iterators

- `std::back_inserter`
  - like `std::inserter` but adds to the end
  - calls `push_back`
- `std::front_inserter`

Iterators

Other output iterators

- `std::back_inserter`
- `std::front_inserter`
  - Like `std::back_inserter` but adds to the front
  - calls `push_front`

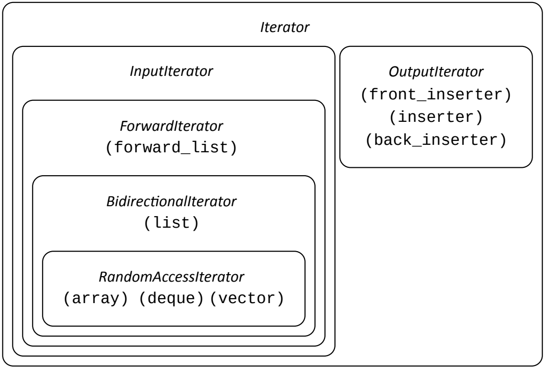
Iterators

Other output iterators

- `std::back_inserter`
- `std::front_inserter`
- These only need to know the container, since their insertion positions are fixed

Iterators

Iterator hierarchy



Iterators

What will be printed?

```
int main()
{
    std::vector<int> v {1, 3};
    *std::back_inserter(v)++ = 7;

    int value { ++(*v.begin()) };
    std::inserter(v, v.begin() + 1) = value;
    for (int i : v)
    {
        std::cout << i << " ";
    }
}
```

LINKÖPING UNIVERSITY

Associative Containers

`std::set`

```
std::set<int> set{};
```

Associative Containers

`std::set`

```
{}
```

```
std::set<int> set{};
```

Associative Containers

`std::set`

{}

```
set.insert(4);
```

Associative Containers

`std::set`

{4}

```
set.insert(4);
```

Associative Containers

`std::set`

{4}

```
set.insert(3);
```

Associative Containers

`std::set`

{3, 4}

```
set.insert(3);
```

Associative Containers

`std::set`

{3, 4}

```
set.insert(5);
```

Associative Containers

`std::set`

{3, 4, 5}

```
set.insert(5);
```

Associative Containers

`std::set`

$$\{3,\ 4,\ 5\}$$

```
set.insert(1);
```

Associative Containers

`std::set`

{1, 3, 4, 5}

```
set.insert(1);
```

Associative Containers

`std::set`

{1, 3, 4, 5}

```
set.insert(2);
```

Associative Containers

`std::set`

{1, 2, 3, 4, 5}

```
set.insert(2);
```

Associative Containers

`std::set`

{1, 2, 3, 4, 5}

```
set.erase(3);
```

Associative Containers

`std::set`

{1, 2, 4, 5}

```
set.erase(3);
```

Associative Containers

`std::set`

- `std::set` contains a *set* of unique values
- requires that the data type of the elements are *comparable*
- will iterate through the elements in sorted order
- is represented as a binary search tree

Associative Containers

`std::set`

- insertion: $O(\log n)$
- deletion: $O(\log n)$
- lookup: $O(\log n)$

Associative Containers

Example

```
#include <set>
// ...
int main()
{
  std::set<std::string> words{};
  std::string str;
  while (cin >> str)
  {
    set.insert(str);
  }
  for (auto const& word : words)
  {
    cout << word << endl;
  }
}
```

Associative Containers

`std::map`


```
std::map<std::string, int> map{};
```

Associative Containers

`std::map`


```
map["c"] = 3;
```

Associative Containers

`std::map`

"c"	3

```
map["c"] = 3;
```

Associative Containers

`std::map`

"c"	3

```
map["a"] = 1;
```

Associative Containers

`std::map`

"a"	1
"c"	3

```
map["a"] = 1;
```

Associative Containers

`std::map`

"a"	1
"c"	3

```
map["d"] = 4;
```

Associative Containers

`std::map`

"a"	1
"c"	3
"d"	4

```
map["d"] = 4;
```

Associative Containers

`std::map`

"a"	1
"c"	3
"d"	4

```
map["b"] = 2;
```

Associative Containers

`std::map`

"a"	1
"b"	2
"c"	3
"d"	4

```
map["b"] = 2;
```

Associative Containers

`std::map`

- `std::map` associates a *key* with a *value*
- keys must be unique and comparable
- will iterate through the key-value pairs in sorted order (according to the key)
- both lookup and insertion can be done with `operator[]`
- implemented as a binary search tree

Associative Containers

`std::map`

- insertion: $O(\log n)$
- deletion: $O(\log n)$
- lookup: $O(\log n)$

Associative Containers

Example

```
#include <map>
// ...
int main()
{
  std::map<std::string, int> words{};
  std::string str;
  while (cin >> str)
  {
    words[str]++;
  }
  for (std::pair<std::string, int> const& p : words)
  {
    cout << p.first << ": " << p.second << endl;
  }
}
```

Associative Containers

Variants

- `std::set` and `std::map` are the base associative containers
- but there are several variations of these containers that can be used
- it is important to choose the appropriate variant

Associative Containers

Variants

- `multi*`
- `unordered_*`

Associative Containers

Variants

- `multi*`
  - `std::multiset`
  - `std::multimap`
- `unordered_*`

Associative Containers

Variants

- `multi*`
- `unordered_*`
  - `std::unordered_set`
  - `std::unordered_map`
  - `std::unordered_multiset`
  - `std::unordered_multimap`

Associative Containers

`std::multiset` and `std::map`

- just like `std::set` and/or `std::map` but with one exception;
- possible to store multiple duplicates of the same key
- can be likened to a list where elements are sorted by the key
- In general these variants are more costly compared to their normal counterparts

Associative Containers

unordered variants

- works like the other associative containers;
- but the elements are not sorted
- these are usually implemented as hash tables
- often a bit faster than the other variants since there are less constraints on the implementation
- **Note:** the order is not defined, so assume *nothing* about the order

LINKÖPING UNIVERSITY

Container Adaptors

Adaptors

- adaptors are wrappers around other containers
- adaptors exposes specific interfaces
- but they are not containers by themselves

Container Adaptors

`std::stack`

```
template <typename T,
          typename Container = std::deque<T>>
class stack;
```

```
std::stack<int> st{};
st.top();  // top of stack
st.push(); // push to stack
st.pop();  // pop the stack
```

Container Adaptors

`std::stack`

`std::stack` can be wrapped around any container that has the following member functions:

- `back()`
- `push_back()`
- `pop_back()`

Container Adaptors

`std::queue`

```
template <typename T,
          typename Container = std::deque<T>>
class queue;
```

```
std::queue<int> q{};
q.front(); // front of the queue
q.back();  // back of the queue
q.push();  // add element to back of queue
q.pop();   // pop first element of the queue
```

Container Adaptors

`std::queue`

`std::queue` can be wrapped around any container that has the following member functions:

- `back()`
- `front()`
- `push_back()`
- `pop_front()`

Container Adaptors

`std::priority_queue`

```
template <typename T,
          typename Container = std::vector<T>,
          typename Compare = std::less<T>>
class priority_queue;
```

```
std::priority_queue<int> pq{};
pq.top();  // get the largest value
pq.push(); // add an element
pq.pop();  // extract the largest value
```

Container Adaptors

`std::priority_queue`

- represents a (min- or max) heap
- stores it in some array-like container
- we can supply some custom comparator

Container Adaptors

`std::priority_queue`

`std::priority_queue` can be wrapped around any container which fullfill the following requirements:

- the container has *RandomAccessIterators*
- It must provide the following functions:
  - `front()`
  - `push_back()`
  - `pop_back()`

Container Adaptors

Example

```
int main()
{
  std::priority_queue<float, std::greater<float>> q{};

  float value;
  while (cin >> value)
  {
    q.push(value);
  }

  float sum{0.0};
  while (!q.empty())
  {
    sum += q.top();
    q.pop();
  }

  cout << sum << endl;
}
```

Container Adaptors

Explanation

- floating point addition is more precise for smaller values
- so if we add the smallest elements first we will get better accuracy
- use the `std::greater` comparator to make our queue a min-heap
- that way, we will add the numbers in ascending order

LINKÖPING UNIVERSITY

Lambda Functions

Possible implementation of `std::less`

```cpp
template <typename T>
struct less
{
  bool operator()(T const& lhs,
                  T const& rhs)
  {
    return lhs < rhs;
  }
};
```

```cpp
int main()
{
  less<int> obj{};

  // we can use the function call
  // operator  to treat this object
  // as a function
  cout << obj(1, 2) << endl;
}
```

Lambda Functions

Possible implementation of `std::less`

- `less` is a class-type
- `less` defines the function call operator `operator()`
- therefore all instances of `less` are both objects and functions at the same time
- these types of classes are called *function objects* or *functors*

Lambda Functions

First-class functions

```
template <typename Function>
auto perform(Function f) -> decltype(f())
{
  return f();
}
```

Lambda Functions

First-class functions

```
struct my_function
{
  int operator()()
  { return 1; }
};

int main()
{
  my_function f{};
  perform(f);
}
```

Lambda Functions

First-class functions

- Function-objects allows us to treat functions as data
- we can pass function-objects as parameters
- with this we can create highly customizable code

Lambda Functions

Example

```
template <typename Container,
          typename Compare>
bool is_sorted(Container const& c,
               Compare const& comp)
{
  auto it{c.begin()};
  auto prev{it++};
  for (; it != c.end(); ++it)
  {
    if (!comp(*prev, *it))
      return false;
    prev = it;
  }
  return true;
}
```

```
int main()
{
  std::vector<int> v{1,2,3,4};
  std::deque<int>  d{3,2,1,0};

  std::less<int>    lt{};
  std::greater<int> gt{};

  cout << is_sorted(v, lt);
  cout << is_sorted(d, gt);
}
```

Lambda Functions

Example

- The sorted function uses `comp` to compare each element with the one before
- note that we can pass in any function object as second parameter
- as long as `comp` is callable, takes two parameters and returns a `bool` this will work
- will work for both function objects and normal functions
- we can even define our own function object

Lambda Functions

Lambda expressions

```
std::vector<int> v{10, -1, 12};

is_sorted(v, [](int a, int b) -> bool
             {
               return abs(a - 10) < abs(b - 10);
             });
```

Lambda Functions

Lambda expressions

```
[](int a, int b) -> bool
{
  return abs(a - 10) < abs(b - 10);
}
```

```
struct my_lambda
{
  bool operator()(int a, int b)
  {
    return abs(a - 10) < abs(b - 10);
  }
};
```

Lambda Functions

Lambda expressions

- Lambda functions:

```
[ captures ] ( parameters ) -> result { body; }
```

- essentially short-hand notation for generating function objects
- useful when creating functions that are passed as parameters

Lambda Functions

Captures

```
std::vector<int> v{10, -1, 12};
int x{10};

auto comp{[x](int a, int b) -> bool
          {
            return abs(a - x) < abs(b - x);
          }};

is_sorted(v, comp);
```

Lambda Functions

Captures

```
[x](int a, int b) -> bool
{
  return abs(a - x) < abs(b - x);
}
```

```
struct my_lambda
{
  my_lambda(int x) : x{x} { }
  bool operator()(int a, int b)
  {
    return abs(a - x) < abs(b - x);
  }
private:
  int const x;
};
```

Lambda Functions

Captures

```
[&x](int a, int b) -> bool
{
  return abs(a - x) < abs(b - x);
}
```

```
struct my_lambda
{
  my_lambda(int& x) : x{x} { }
  bool operator()(int a, int b)
  {
    return abs(a - x) < abs(b - x);
  }
private:
  int& x;
};
```

Lambda Functions

Captures

```
[x = 10](int a, int b) -> bool
{
  return abs(a - x) < abs(b - x);
}
```

```
struct my_lambda
{
  my_lambda() : x{10} { }
  bool operator()(int a, int b)
  {
    return abs(a - x) < abs(b - x);
  }
private:
  int const x;
};
```

Lambda Functions

Captures

```
[x, &y, z = 10](/* function parameters */)
{
  // ...
}
```

Lambda Functions

Captures

- It is possible to make extra variables available inside lambda expressions
- These extra variables (non-parameters) are said to be *captured* inside the lambda
- They can either be copies or references of variables declared outside the lambda
- They can also be completely new variables that attain its current value between calls

Lambda Functions

`mutable`

```
int x{};
auto f = [x]() { x = 1; };
```

Lambda Functions

`mutable`

```
int x{};
auto f = [x]() mutable { x = 1; };
```

Lambda Functions

`mutable`

- Everything captured by-value in lambdas will be `const`
- Including variables defined inside the lambda
- To make them non-`const` we have to declare the lambda as `mutable`

Lambda Functions

Special captures

```
int global{1};
int main()
{
  int x{2};
  int y{3};
  auto f{[&]()
         {
           return x + y + global;
         }};
  f(); // will return 6
  y = -3;
  f(); // will return 0
}
```

Lambda Functions

Special captures

```
int global{1};
int main()
{
  int x{2};
  int y{3};
  auto f{[=]()
         {
           return x + y + global;
         }};
  f(); // will return 6
  y = -3;
  f(); // will return 6
}
```

Lambda Functions

Special captures

- Capture everything as reference:

```
[&] ( parameters ) -> result { body }
```

- Capture everything as a copy:

```
[=] ( parameters ) -> result { body }
```

- Both of these act as if they capture every variable accessible in the code at the point of definition
- However; in reality they will capture only those variables that are used inside the body

Lambda Functions

Mixing captures

```
[=, &x](/* parameters */)
{
  // ...
}
```

Capture everything as copy, except x; capture x as reference.

Lambda Functions

What will be printed?

```
int main()
{
  auto f = [n = 0]() mutable { return n++; };
  auto g = f;
  cout << f() << ' ';
  cout << f() << ' ';
  cout << g() << endl;
}
```

www.liu.se

LINKÖPING
UNIVERSITY