TDDE18 & 726G77

Standard library

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Containers

Introduction

- Sequence containers
- Sequence adaptors
- Associative containers

Containers

Introduction

- Sequence containers
    - Store values of the same type in a given sequence
    - Normally we use an index to retrieve the values
- Sequence adaptors
- Associative containers

Containers

Introduction

- Sequence containers
- Sequence adaptors
  - Adapted interface for a given sequence container
  - Represents ADT:s such as stacks, queues and priority queues
- Associative containers

Containers

Introduction

- Sequence containers
- Sequence adaptors
- Associative containers
  - Store values associated with given keys
  - Values must be of the same data type
  - Keys must be of the same data type
  - Use the key to retrieve corresponding value

Containers

Sequence container

- `vector`
  - Store values in consecutive memory
  - Grows to accommodate its content
  - The most common containers
- `array`
- `deque`
- `list`
- `forward_list`

Containers

Sequence container

- `vector`
- `array`
  - Store values in consecutive memory
  - Fixed size that is known during compilation
  - Is more effective (both in memory and speed) than `vector`
- `deque`
- `list`
- `forward_list`

Containers

Sequence container

- `vector`
- `array`
- `deque`
  - **D**ouble-**e**nded **que**ue
  - Does **not** store values in consecutive-memory
  - Very good to use if you want to add/remove values in the beginning AND end
- `list`
- `forward_list`

Containers

Sequence container

- `vector`
- `array`
- `deque`
- `list`
  - A doubly-linked list (previous pointer as well as a next pointer)
  - Does **not** store values in consecutive-memory
  - Very good to use if you want to add/remove values in the beginning
  - Can step backwards and forward in the list
- `forward list`

Containers

Sequence container

- `vector`
- `array`
- `deque`
- `list`
- `forward_list`
    - A singly-linked list
    - Does **not** store values in consecutive-memory
    - Very good to use if you want to add/remove values in the beginning
    - Can only step forward in the list

Containers

Sequence container

```
#include <vector>
#include <array>
#include <deque>
#include <list>
#include <forward_list>
int main()
{
  // likewise for list, forward_list and deque
  std::vector<int>   v {1, 2, 3};

  // we have to specify a size for array
  std::array<int, 3> a {1, 2, 3};
}
```

Containers

Sequence adapters

- `stack`
- `queue`
- `priority_queue`

Containers

Sequence adapters

- `stack`
  - Usually built on top of `deque`
  - Can only access/remove the last inserted value
- `queue`
- `priority_queue`

Containers

Sequence adapters

- `stack`
- `queue`
    - First-In First-Out, a queue
    - Usually built on top of `deque`
    - Can only add values to the end and remove from the beginning
- `priority_queue`

Containers

Sequence adapters

- `stack`
- `queue`
- `priority_queue`
  - A queue that is ordered after a given priority
  - Can only access/remove the value with the highest priority
  - Values are always sorted by their priority

Containers

Sequence adapters

```
#include <stack>
#include <queue>
#include <priority_queue>
int main()
{
  // likewise for all adapters
  std::stack<int> s1 {};

  // we can change which container it uses
  std::stack<int, std::vector<int>> s2 {};
}
```

Containers

Associative containers

- `map`
- `set`
- `multi*`
- `unordered_*`

Containers

Associative containers

- `map`
  - Associates a value with a key
  - Requires the keys to be comparable with `operator<`
  - Sorted by the keys
  - Each key can only occur once
- `set`
- `multi*`
- `unordered_*`

Containers

Associative containers

- `map`
- `set`
  - Like `map` but only store keys
  - Guarantees that all inserted values are unique and sorted
- `multi*`
- `unordered_*`

Containers

Associative containers

- `map`
- `set`
- `multi*`
  - `multimap` and `multiset`
  - Like `map` and `set` respectively, but the keys doesn't have to be unique anymore
- `unordered_*`

Containers

Associative containers

- `map`
- `set`
- `multi*`
- `unordered_*`
    - `unordered_map`, `unordered_multimap`, `unordered_set` and `unordered_multiset`,
    - The keys are no longer sorted
    - The keys doesn't have to be comparable anymore (however they need to be *hashable*)

Containers

Associative containers

```
#include <map>
#include <set>
#include <string>
int main()
{
  // associates "a" with 1 and "b" with 2
  std::map<std::string, int> m { {"a", 1},
                                 {"b", 2} };
  std::set<double> s { 1.0, 3.0, -1.0 };
}
```

LINKÖPING UNIVERSITY

Iterators

Iteration

- We want to be able to loop through our containers
- It would be nice if we could do it the same way for all containers
- The problem is that containers doesn’t always have the same way of accessing values
- Therefore we have to generalize our understanding of looping through containers

Iterators

Iteration

```
int main()
{
  vector<int> v {1, 2, 3};
  for (int i{0}; i < v.size(); ++i)
  {
    cout << v[i] << endl;
  }
}
```

Iterators

Iteration

```
int main()
{
  set<int> v {1, 2, 3};
  for (int i{0}; i < v.size(); ++i)
  {
    cout << v[i] << endl; // doesn't work
  }
}
```

Iterators

Iteration

```
int main()
{
  vector<int> v {1, 2, 3};
  for (int e : v)
  {
    cout << e << endl;
  }
}
```

Iterators

Iteration

```
int main()
{
  set<int> v {1, 2, 3};
  for (int e : v)
  {
    cout << e << endl; // works!
  }
}
```

Iterators

Range-based for-loop

- It is possible to create your own containers
- How can we enable this general loop to work for our custom-made container?
- This is where the concept of iterators comes in!

Iterators

Range-based for-loop

```
for (int e : v)
{
  cout << e << endl;
}
```

Iterators

Range-based for-loop

```
using iterator = std::vector<int>::iterator;

for (iterator it{v.begin()}; it != v.end(); ++it)
{
  cout << *it << endl;
}
```

Iterators

Range-based for-loop

A container can be looped through if:

- There is an inner class called `iterator`
- There are member functions `begin` and `end`, both of which return `iterator` objects
- `iterator` has defined `operator++`, `operator*`, `operator==` and `operator!=`

Iterators

Iterators

- Iterators are generalized pointers
- Represent a general way of iterating over containers
- Points to a value in the container
- Possible to access values with `operator*`
- Go to the next element with `operator++`

Iterators

Iterators

```
vector<int> v{1,2,3};
```

Iterators

Iterators

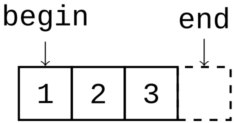
```
vector<int> v{1,2,3};
```

1	2	3

Iterators

Iterators

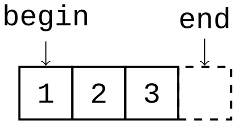
```
vector<int> v{1,2,3};
```



Iterators

Iterators

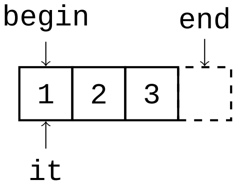
```
vector<int>::iterator it{v.begin()};
```



Iterators

Iterators

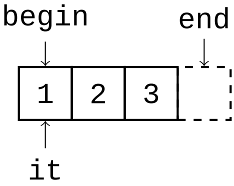
```
vector<int>::iterator it{v.begin()};
```



Iterators

Iterators

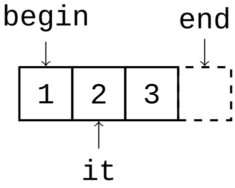
```
++it;
```



Iterators

Iterators

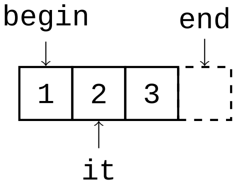
```
++it;
```



Iterators

Iterators

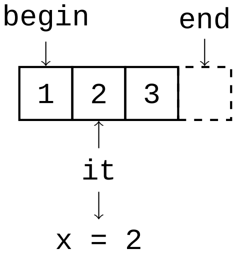
```
int x{*it};
```



Iterators

Iterators

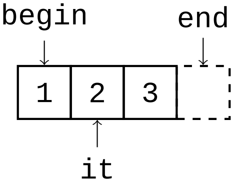
```
int x{*it};
```



Iterators

Iterators

```
int x{*it};
```

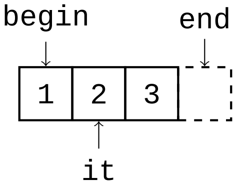


x = 2

Iterators

Iterators

```
++it;
```

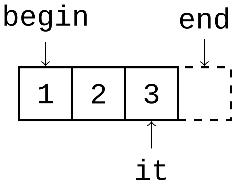


x = 2

Iterators

Iterators

```
++it;
```

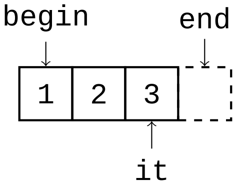


x = 2

Iterators

Iterators

```
*it = 4;
```

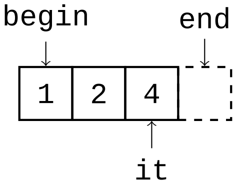


x = 2

Iterators

Iterators

```
*it = 4;
```

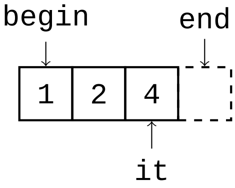


x = 2

Iterators

Iterators

```
++it;
```

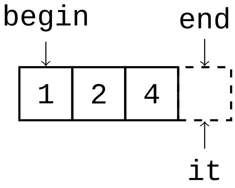


x = 2

Iterators

Iterators

```
++it;
```

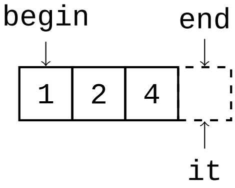


x = 2

Iterators

Iterators

```
it == v.end();
```



x = 2

Iterators

- It is important that the `end`-iterator doesn’t point to the last value
- If it does we will miss the last element in the container since the loop is ended whe `it == v.end()`
- We must be able to uniquely identify that we have iterated through all elements
- Therefore we think of the `end`-iterator as a pointer to the value after the last one

Iterators

Iterator categories

There are different types of iterators, these are:

- Input
  - Can ready already existing values in a container
- Output
- Forward
- Bidirectional
- Random Access

Iterators

Iterator categories

There are different types of iterators, these are:

- Input
- Output
  - Can add new values in a container
- Forward
- Bidirectional
- Random Access

Iterators

Iterator categories

There are different types of iterators, these are:

- Input
- Output
- Forward
  - Can read/overwrite existing values in a container
  - Can step forward in the container
  - Is also an Input iterator
- Bidirectional
- Random Access

Iterators

Iterator categories

There are different types of iterators, these are:

- Input
- Output
- Forward
- Bidirectional
  - Is a forward iterator
  - But can also step backwards with `operator--`
- Random Access

Iterators

Iterator categories

There are different types of iterators, these are:

- Input
- Output
- Forward
- Bidirectional
- Random Access
  - Can access arbitrary elements in the container with `operator+`
  - Is also a Bidirectional iterator

Iterators

Iterator categories

The following table demonstrates which operations are possible for the different categories

	Category				
Operationer	**Input**	**Output**	**Forward**	**Bidirectional**	**Random Access**
==, !=	✓	✓	✓	✓	✓
*, ->	Read	Write	Read/Write	Read/Write	Read/Write
++	✓	✓	✓	✓	✓
–	-	-	-	✓	✓
+, +=, -, -=	-	-	-	-	✓
<, <=, >, >=	-	-	-	-	✓
i[n]	-	-	-	-	✓

LINKÖPING UNIVERSITY

Standard Library

What is the Standard Library?

- Available for everyone
  - Same behaviour regardless of computer and operating system
  - Is included with the compiler
  - ISO C++ requires everything to be implemented
- Solves common problems
- Components
- Effective

Standard Library

What is the Standard Library?

- Available for everyone
- Solves common problems
    - Reinventing the wheel takes time
    - There are problems all programmers face
    - Should be widely applicable
- Components
- Effective

Standard Library

What is the Standard Library?

- Available for everyone
- Solves common problems
- Components
  - Don’t pay for what you don’t use
  - Import only the parts that you need
  - Everything is compatible with each other
- Effective

Standard Library

What is the Standard Library?

- Available for everyone
- Solves common problems
- Components
- Effective
  - The people that implement the library know what they are doing
  - Everything is highly optimized
  - It is not your responsibility to make sure it works

Standard Library

STL

Standard **T**emplate **L**ibrary

Standard Library

Design goals

- Should be as general as possible

Standard Library

Design goals

- Should be as general as possible
- Solves common problems

Standard Library

Design goals

- Should be as general as possible
- Solves common problems
- The common case should be easy

Standard Library

Design goals

- Should be as general as possible
- Solves common problems
- The common case should be easy
- Must work with the users code

Standard Library

Design goals

- Should be as general as possible
- Solves common problems
- The common case should be easy
- Must work with the users code
- Should be effective enough to replace hand-written alternatives

Standard Library

Design goals

- Should be as general as possible
- Solves common problems
- The common case should be easy
- Must work with the users code
- Should be effective enough to replace hand-written alternatives
- Should have robust error handling

Standard Library

Components

- Algoritmer
  - General functions to solve common problems
  - Perform operations on containers
  - Uses iterators as an interface against containers
  - Optimized for speed and memory usage
- Containers
- Iterators
- Others

Standard Library

Components

- Algoritmer
- Containers
  - Different ways to structure data
  - Based on common abstractions
  - We shouldn't have to know how it works
- Iterators
- Others

Standard Library

Components

- Algoritmer
- Containers
- Iterators
    - Interface for iterating over data
    - Used as an abstraction for containers
- Others

Standard Library

Components

- Algoritmer
- Containers
- Iterators
- Others
  - General functions and classes
  - Solves various problems
  - Should be usable for as many types as possible

LINKÖPING UNIVERSITY

Algorithms

Why?

- Standard algorithms allows us to communicate clearly what the code does
- Other programmers understand standard algoritms fast, while it requires more time to understand hand-written solutions
- Less steps for us to think about

Algorithms

What does this code do?

```
std::vector<int> v { 5, -2, 8, 4, 7 };

auto it{v.begin()};
for (; it != v.end(); ++it)
{
  if (*it == 4)
    break;
}
if (it == v.end())
{
  // found nothing
}
```

Algorithms

What does this code do?

```
std::vector<int> v { 5, -2, 8, 4, 7 };
auto it {std::find(v.begin(), v.end(), 4)};
if (it == v.end())
{
  // found nothing
}
```

Algorithms

What does this code do?

Which version is easier to understand?

Algorithms

What does this code do?

- Algorithms makes code more readable,
- we don’t have to write the same code over and over again,
- you can think on a higher level,
- no need to think (as much) about optimality.

Algorithms

What does this code do?

```
std::vector<int> v {1, 2, 3, 1, 4, 1};
int result {0};
for (auto it{v.begin()}; it != v.end(); ++it)
{
  if (*it == 1)
    result++;
}
```

Algorithms

What does this code do?

```
std::vector<int> v {1, 2, 3, 1, 4, 1};
int result {std::count(v.begin(), v.end(), 1)};
```

Algorithms

Which algorithms are there?

- There are over 100 different algorithms available
- For each version of C++ new algorithms are added
- A complete list is available here:
  https://en.cppreference.com/w/cpp/algorithm

Algorithms

Modifying algorithms

```
std::vector<int> v {1, 2, 3, 1, 4, 1};
```

Algorithms

Modifying algorithms

```
std::vector<int> v {1, 2, 3, 1, 4, 1};
```

1	2	3	1	4	1

Algorithms

Modifying algorithms

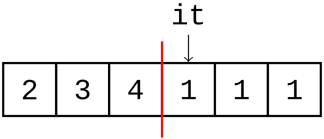
```
auto it {std::remove(v.begin(), v.end(), 1)};
```

1	2	3	1	4	1

Algorithms

Modifying algorithms

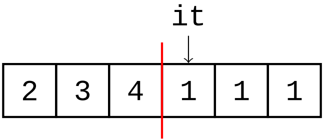
```
auto it {std::remove(v.begin(), v.end(), 1)};
```



Algorithms

Modifying algorithms

```
v.erase(it, v.end());
```



Algorithms

Modifying algorithms

```
v.erase(it, v.end());
```

2	3	4

Algorithms

Modifying algorithms

```
std::vector<int> v {1, 2, 3, 1, 4, 1};
v.erase(std::remove(v.begin(), v.end(), 1),
        v.end());
```

Algorithms

Modifying algorithms

- Some algorithms *remove* values
- There is no functionality in iterators that allows us to actually remove (or insert) values
- What happens instead is that the algorithm moves all *removed* values to the end of the container and return an iterator to the first of the removed values
- Then it is up to the developer to actually remove these values (most often with `erase`)

Algorithms

Copying

```
// command-line argument
std::vector<string> args {argv, argv + argc};

// empty vector
std::vector<string> v {};

// copy all arguments to the empty vector
std::copy(args.begin(), args.end(), v.begin());
```

Algorithms

Copying

- The empty vector have no values
- These iterators can only read and overwrite existing values
- Therefore we try to copy values from `args` to a vector (`v`) that doesn't have any elements that can be written to

Algorithms

Copying

```
// command-line arguments
std::vector<string> args {argv, argv + argc};

// vector with the right amout on elements
std::vector<string> v (args.size());

// copy all arguments to the empty vector
std::copy(args.begin(), args.end(), v.begin());
```

Algorithms

Iterator kategori

Some algorithms don’t work with all containers

Algorithms

Iterator kategori

```
std::vector<int> vals{1, 2, 7, 4, -1};
std::sort(vals.begin(), vals.end());
```

Algorithms

Iterator kategori

```
std::list<int> vals{1, 2, 7, 4, -1};
std::sort(vals.begin(), vals.end());
```

Algorithms

Iterator kategori

```
std::list<int> vals{1, 2, 7, 4, -1};
std::sort(vals.begin(), vals.end());
```

Doesn't work!

Algorithms

Iterator category

- Why won't it work for `std::list`?
- It is because sorting data requires the use of jumping between arbitrary values
- Meaning, it requires `RandomAccessIterator` (the one with `operator+`)
- `std::list` only have `BidirectionalIterator`

Algorithms

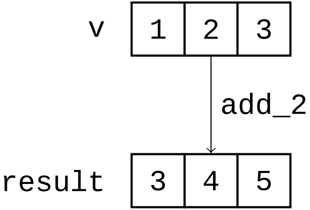
`std::transform`

```
int add_2(int n)
{
  return n + 2;
}

int main()
{
  std::vector<int> v {1, 2, 3};
  std::vector<int> result (v.size());
  std::transform(v.begin(), v.end(),
                 result.begin(), add_2);
}
```

Algorithms

`std::transform`



Algorithms

`std::transform`

- `std::transform` work like `std::copy`...
- But it will first apply the given function on the values
- In this case that means that we copy each value from `v`, add `2` to the value, and then placing it into `result`

Algorithms

`std::transform`

```
int add_2(int n)
{
  return n + 2;
}

int main()
{
  std::vector<int> v {1, 2, 3};
  std::transform(v.begin(), v.end(),
                 v.begin(), add_2);
}
```

Algorithms

`std::transform`

v	1	2	3

Algorithms

`std::transform`

v	4	5	6

Algorithms

`std::transform`

- You can also use `std::transform` to write into the same vector from which you read the values
- This is actually very common
- Of course it requires that the return type from the function is the same as the values in the vector

Algorithms

`std::transform`

- In many cases you only need a specific function for one algorithm call and then never again
- This can lead to many functions in the code that are only used once (which will cause clutter)
- It would be nice if we could create temporary functions that are created together with our `std::tranform` call...

Lambda Functions

Temporary functions

- A lambda function is an expression that creates a temporary function
- They allow us to create and use functions without giving them a name
- They are also more general than normal functions

Lambda Functions

Temporary functions

```
[](int n) -> int { return n + 2; }
```

Lambda Functions

Temporary functions

```
[](int n) -> int { return n + 2; }
```

`[]`	capture
`(int n)`	inparameter
`-> int`	return type
`{ return n + 2; }`	body

Lambda Functions

Return type

```
[](int n) -> int { return n + 2; }
```

Lambda Functions

Return type

```
[](int n) -> auto { return n + 2; }
```

Lambda Functions

Return type

- We can have `auto` as return type
- But the nice thing with lambda functions are that this happens automatically if we don't specify a return type

Lambda Functions

Return type

```
[](int n) { return n + 2; }
```

Lambda Functions

`std::transform`

```
std::vector<int> v {1, 2, 3};
std::transform(v.begin(), v.end(), v.begin(),
               [](int n) { return n + 2; });
```

Lambda Functions

Lambda Functions and STL

- There are many algorithms that take functions as arguments
- Increadibly common that we use lambda functions in this context
- The code becomes shorter and easier to read
- The user doesn’t have find the function declaration and can see directly what the function does

Lambda Functions

Give lambda functions a name

```
auto add_2 = [](int n) { return n + 2; };
std::transform(v.begin(), v.end(),
               v.begin(), add_2);
```

Lambda Functions

Give lambda functions a name

- This way we can keep the abstraction of treating our lambda as a function
- But we don't have to force the reader to look for the functions definition far way, because it is right above the algorithm call
- This also means that `add_2` is a variable, not a function since it only exists locally in the current scope.

Lambda Functions

What is a lambda function?

```
[](int n)
{
  return n + 2;
}
```

```
struct My_Lambda
{
  auto operator()(int n)
  {
    return n + 2;
  }
};
```

Lambda Functions

`operator()`

- `operator()` is called the *function-call operator*
- If we have an object `obj` that is of a type where `operator()` is defined, then: `obj(x)` is translated to `obj.operator()(x)`
- All classes that defines `operator()` are called *function objects*

Lambda Functions

What is a lambda function?

```
auto add_2 {
  [](int n)
  {
    return n + 2;
  }};
```

```
My_Lambda add_2 {};
```

Lambda Functions

Capture

```
int x {2};
std::vector<int> v {1, 2, 3};
std::transform(v.begin(), v.end(), v.begin(),
               [](int n) { return n + x; });
```

Lambda Functions

Capture

```
int x {2};
std::vector<int> v {1, 2, 3};
std::transform(v.begin(), v.end(), v.begin(),
               [](int n) { return n + x; });
```

Doesn't work!

Lambda Functions

Capture

```
int x {2};
std::vector<int> v {1, 2, 3};
std::transform(v.begin(), v.end(), v.begin(),
               [x](int n) { return n + x; });
```

Lambda Functions

Capture

- Within `[]` one can specify which variables should be available inside the lambda function
- This is called the lambda functions *capture*
- This will create a local copy of the variables (these copies are only available inside the lambda)

Lambda Functions

Capture

```
[x](int n)
{
  return n + x;
}
```

```
struct My_Lambda
{
  My_Lambda(int x)
    : x{x} { }
  auto operator()(int n)
  {
    return n + x;
  }
private:
  int x;
};
```

Lambda Functions

Capture

```
int x {2};
auto add_x {
  [x](int n)
  {
    return n + x;
  }};
```

```
int x {2};
My_Lambda add_x {x};
```

Lambda Functions

Capture

```
int x {2};
auto add_x = [x](int n) { return n + x; };
cout << add_x(5) << endl; // 7
x = 3;
cout << add_x(5) << endl; // 7
```

Lambda Functions

Capture

```cpp
int x {2};
auto add_x = [&x](int n) { return n + x; };
cout << add_x(5) << endl; // 7
x = 3;
cout << add_x(5) << endl; // 8
```

Lambda Functions

Capture

- If we add a & before the variables name in the caputre then it will be bound as a reference instead

Lambda Functions

Capture

```
[&x](int n)
{
  return n + x;
}
```

```
struct My_Lambda
{
  My_Lambda(int& x)
    : x{x} { }
  auto operator()(int n)
  {
    return n + x;
  }
private:
  int& x;
};
```

Lambda Functions

Capture all

```
int x{2};
int y{3};

auto f = [&](int n) { return y*n + x; };

std::vector<int> v {1, 2, 3};
std::transform(v.begin(), v.end(),
               v.begin(), f);

// v == {5, 8, 11}
```

Lambda Functions

Capture all

- If we only write `[&]` then we capture *all* variables that are available at the point of definition for the lambda
- In reality it only capture those variable that are used inside the lambda function
- Captures everything as a reference

Lambda Functions

Function objects in STL

```
std::vector<int> v {4, 6, 3, 7, 1};
std::sort(v.begin(), v.end());

// v == {1, 3, 4, 6, 7}
```

Lambda Functions

Function objects in STL

```
std::vector<int> v {4, 6, 3, 7, 1};
std::sort(v.begin(), v.end(),
          [](int x, int y) { return x > y; });

// v == {7, 6, 4, 3, 1}
```

Lambda Functions

Function objects in STL

```
std::vector<int> v {4, 6, 3, 7, 1};
std::sort(v.begin(), v.end(), greater<int>{});

// v == {7, 6, 4, 3, 1}
```

Lambda Functions

Function objects in STL

- There are a couple of builtin function objects that can be used with algorithms
- Some useful examples: `std::less`, `std::greater`, `std::plus`, etc.
- All these function objects are listed here: https://en.cppreference.com/w/cpp/utility/functional (**Operator function objects**)

LINKÖPING UNIVERSITY

More on iterators

`std::for_each`

```
std::vector<int> v {1, 2, 3, 4};
for (int e : v)
{
  cout << e << ' ';
}
```

More on iterators

`std::for_each`

```
std::vector<int> v {1, 2, 3, 4};
std::for_each(v.begin(), v.end(), [](int e)
{
  cout << e << ' ';
});
```

More on iterators

`std::for_each`

- `std::for_each` is a relic from the old days
- C++ have developed to such a level that `std::for_each` is *almost never* needed
- So avoid `std::for_each` whenever possible

More on iterators

Print a container

```
std::vector<int> v {1, 2, 3, 4};
std::copy(v.begin(), v.end(),
          std::ostream_iterator<int>{cout});
```

More on iterators

Print a container

Prints:

```
1234
```

More on iterators

Print a container

```
std::vector<int> v {1, 2, 3, 4};
std::copy(v.begin(), v.end(),
          std::ostream_iterator<int>{cout, " "});
```

More on iterators

Print a container

Prints:

```
1 2 3 4
```

More on iterators

`std::ostream_iterator`

- Is an `OutputIterator`
- Given `std::ostream_iterator<int> it{cout}` the expression `*it = 5` will print `5` to `cout`
- `++it` and `it++` does nothing

More on iterators

Read from a stream

```
std::vector<int> v {};
int x;
while (std::cin >> x)
{
  v.push_back(x);
}
```

More on iterators

Read from a stream

```
std::vector<int> v {
  std::istream_iterator<int>{cin},
  std::istream_iterator<int>{}
};
```

More on iterators

Read from a stream

- `std::vector` have a constructor that copies values from a pair of iterators
- Given a start and a end-iterator this constructor will copy each of these values into the vector

More on iterators

Read from a stream

- `std::istream_iterator` is an *InputIterator*
- Given `std::istream_iterator<int> it {cin}` the expression `*it` will return the latest read value from `cin`
- `it++` or `++it` will read the next value from `cin`
- Can be used as if `cin` was a container
- The default-constructor creates an *end*-iterator

More on iterators

Output iterator

```
std::vector<int> v {};
std::transform(std::istream_iterator<int>{cin},
               std::istream_iterator<int>{},
               v.begin(),
               [](int e) { return 2 * e; });
```

More on iterators

Output iterator

```
std::vector<int> v {};
std::transform(std::istream_iterator<int>{cin},
               std::istream_iterator<int>{},
               std::back_inserter(v),
               [](int e) { return 2 * e; });
```

More on iterators

Output iterator

- `back_inserter` creates an *OutputIterator* that will call `push_back` everytime we assign to it
- Given `std::vector<int> v` and `auto it{std::back_inserter(v)}` the expression `*it++ = 5` will be equivalent with calling `v.push_back(5)`
- This is very useful together with algorithms such as `std::copy` and `std::transform`

More on iterators

```
std::vector<int> args {};

std::transform(argv + 1, argv + argc,
               std::back_inserter(args),
               [](char const* arg)
               {
                 return std::stoi(arg);
               });

std::sort(args.begin(), args.end(),
          std::greater<int>);

std::copy(args.begin(), args.end(),
          std::ostream_iterator<int>{cout, ", "});
```

More on iterators

If we run:

```
$ ./a.out 7 15 32 1 11
```

More on iterators

It prints:

```
32, 15, 11, 7, 1,
```

Tip for lab 5

look at all member functions of
`std::string`

LINKÖPING UNIVERSITY

Smart pointers

Can we get memory problems here?

```
class My_Class
{
public:
  My_Class(int x, int y);
  ~My_Class()
  {
    delete p1;
    delete p2;
  }
private:
  int* p1;
  int* p2;
};
```

Smart pointers

Can we get memory problems here?

```
int* create(int n)
{
  if (n >= 0)
  {
    return new int{n};
  }
  throw domain_error{"Negative"};
}

My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{create(y)}
{ }
```

Smart pointers

Yes, there are problems!

```
int main()
{
  My_Class c{0, -1};
}
```

Smart pointers

Why?

- When the constructor is aborted the object will be removed without running the destructor
- All data that were allocated before the crash will therefore not be deallocated
- How can we solve this?

Smart pointers

Lösning?

```
My_Class::My_Class(int x, int y) try
  : p1{create(x)}, p2{create(y)}
{ }
catch (domain_errror& e)
{
  delete p1;
}
int main()
{
  My_Class c{-1, 0};
}
```

Smart pointers

Lösning?

```
My_Class::My_Class(int x, int y) try
  : p1{create(x)}, p2{create(y)}
{ }
catch (domain_error& e)
{
  delete p1;
}
int main()
{
  My_Class c{-1, 0};
}
```

Segmentation Fault

Smart pointers

Why?

- Now `p1` will throw an exception
- In the catch-block we are trying to `delete` it, but it has not been allocated
- This gives us a segmentation fault

Smart pointers

Solution?

```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{}
{
  try
  {
    p2 = create(y);
  }
  catch (domain_error& e)
  {
    delete p1;
    throw;
  }
}
```

Smart pointers

Solution?

```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{}
{
  try
  {
    p2 = create(y);
  }
  catch (domain_error& e)
  {
    delete p1;
    throw;
  }
}
```

Works...

Smart pointers

Solution?

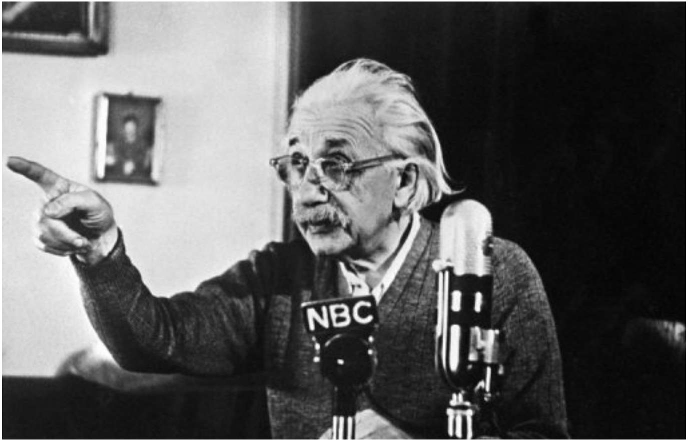
```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{}
{
  try
  {
    p2 = create(y);
  }
  catch (domain_error& e)
  {
    delete p1;
    throw;
  }
}
```

At what cost?

Smart pointers

It would be nice if pointers could deallocate themselves...

Smart pointers



Smart pointers

`std::unique_ptr`

```
int main()
{
  int* p1{new int{5}};
  cout << *p1 << endl;
  {
    int* p2{new int{3}};
    cout << *p2 << endl;
    delete p2;
  }
  delete p1;
}
```

Smart pointers

std::unique_ptr

```
#include <memory>
int main()
{
  std::unique_ptr<int> p1{new int{5}};
  cout << *p1 << endl;
  {
    std::unique_ptr<int> p2{new int{3}};
    cout << *p2 << endl;

  }

}
```

Smart pointers

`std::unique_ptr`

- Is a so-called *smartpointer*
- Is defined in `<memory>`
- Takes responsibility for handling memory
- Represents *ownership*
- Can **not** be copied
- But we can use move-semantics to move the ownership

Smart pointers

`std::unique_ptr`

```
#include <memory>
using namespace std;
int main()
{
  unique_ptr<double> p{}; // nullptr
  p = new double{5.0};
  {
    unique_ptr<double> q{new double{1.0}};
    p = std::move(q);
  }
}
```

Smart pointers

`std::unique_ptr`

- `unique_ptr` removes the old memory when we assign it a new value
- You should *almost never* have to deallocate the memory manually

Smart pointers

Trap

```
#include <memory>
int get(std::unique_ptr<int> p)
{
  return *p;
}
int main()
{
  std::unique_ptr<int> p{new int{5}};
  get(p);
}
```

Smart pointers

Trap

```
#include <memory>
int get(std::unique_ptr<int> p)
{
  return *p;
}
int main()
{
  std::unique_ptr<int> p{new int{5}};
  get(p);
}
```

Compile error

Smart pointers

Trap

```
test.cpp: In function ‘int main’():
test.cpp:9:8: error: use of deleted function ‘std::unique_ptr<_Tp, _Dp>::
unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = int; _Dp = std::
default_delete<int’>]
   get(p);
        ^
In file included from /sw/gcc-7.1.0/include/c++/7.1.0/memory:80:0,
                 from test.cpp:1:
/sw/gcc-7.1.0/include/c++/7.1.0/bits/unique_ptr.h:388:7: note: declared
here
       unique_ptr(const unique_ptr&) = delete;
       ^~~~~~~~~~
test.cpp:2:5: note:   initializing argument 1 of ‘int get(
std::unique_ptr<int’>)
 int get(std::unique_ptr<int> p)
     ^~~
```

Smart pointers

Trap

```
test.cpp: In function ‘int main’():
test.cpp:9:8: error: use of deleted function ‘std::unique_ptr<_Tp, _Dp>::
unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = int; _Dp = std::
default_delete<int’>]
   get(p);
        ^
In file included from /sw/gcc-7.1.0/include/c++/7.1.0/memory:80:0,
                 from test.cpp:1:
/sw/gcc-7.1.0/include/c++/7.1.0/bits/unique_ptr.h:388:7: note: declared
here
       unique_ptr(const unique_ptr&) = delete;
       ^~~~~~~~~~
test.cpp:2:5: note:   initializing argument 1 of ‘int get(
std::unique_ptr<int’>)
 int get(std::unique_ptr<int> p)
     ^~~
```

Smart pointers

Trap

If you see this, then you are trying to copy a `unique_ptr`

Smart pointers

More on `unique_ptr`

```
int main()
{
  std::unique_ptr<std::string> p{};
  // we don't have to worry about the allocations
  p = std::make_unique<std::string>("hello");
  // retrieve a normal pointer
  std::string* ptr{p.get()};
  // access members in the object
  cout << p->size() << endl;
}
```

Smart pointers

`std::make_unique`

- If we use `std::make_unique` rather than `new` we signla to the reader what is going on
- It is clearer that this code doesn't require `delete` if `new` doesn't even occur in the code
- This allows the compiler to reason better about the code and can therefore do potential optimizations that wouldn't be possible otherwise

Smart pointers

`get`

- `std::unique_ptr::get` should only be used when we need *temporary* access to the object
- The pointer that is returned from `get` is a *non-owning* pointer
- This means that you should never call `delete` on it
- If you for some reason want to deallocate the memory, assign `nullptr` to the smartpointer or call `release`

Smart pointers

std::shared_ptr

```
int main()
{
  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
  {
    std::shared_ptr<int> ptr2{ptr1};
    {
      std::shared_ptr<int> ptr3{ptr1};
    }
  }
}
```

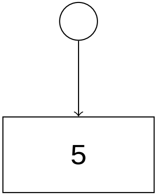
Smart pointers

std::shared_ptr

```
 int main()
 {
>  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

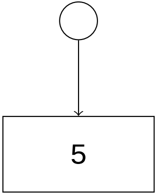
`std::shared_ptr`




```
 int main()
 {
>  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

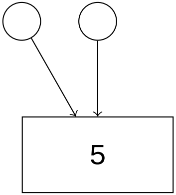
std::shared_ptr




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
>    std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

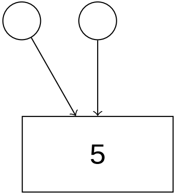
`std::shared_ptr`




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
>    std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

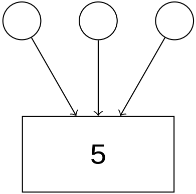
`std::shared_ptr`




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
>      std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

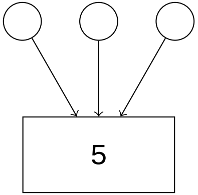
std::shared_ptr




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
>      std::shared_ptr<int> ptr3{ptr1};
     }
   }
 }
```

Smart pointers

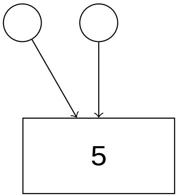
std::shared_ptr




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
>    }
   }
 }
```

Smart pointers

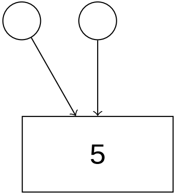
`std::shared_ptr`




```
  int main()
  {
    std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
    {
      std::shared_ptr<int> ptr2{ptr1};
      {
        std::shared_ptr<int> ptr3{ptr1};
>     }
    }
  }
```

Smart pointers

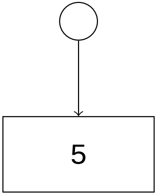
`std::shared_ptr`




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
>  }
 }
```

Smart pointers

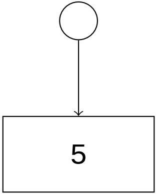
`std::shared_ptr`




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
>  }
 }
```

Smart pointers

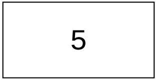
`std::shared_ptr`




```
 int main()
 {
   std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
   {
     std::shared_ptr<int> ptr2{ptr1};
     {
       std::shared_ptr<int> ptr3{ptr1};
     }
   }
>}
```

Smart pointers

`std::shared_ptr`




```
int main()
{
  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
  {
    std::shared_ptr<int> ptr2{ptr1};
    {
      std::shared_ptr<int> ptr3{ptr1};
    }
  }
>}
```

Smart pointers

std::shared_ptr

```
int main()
{
  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
  {
    std::shared_ptr<int> ptr2{ptr1};
    {
      std::shared_ptr<int> ptr3{ptr1};
    }
  }
>}
```

Smart pointers

std::shared_ptr

```
int main()
{
  std::shared_ptr<int> ptr1{std::make_shared<int>(5)};
  {
    std::shared_ptr<int> ptr2{ptr1};
    {
      std::shared_ptr<int> ptr3{ptr1};
    }
  }
}
```

Smart pointers

`std::shared_ptr`

- Represent shared ownership
- The memory is only deallocated when no one is pointing to it
- Costs more than `std::unique_ptr` and normal pointers

Smart pointers

How does this help us?

Smart pointers

Better solution!

```
class My_Class
{
public:
  My_Class(int x, int y);
  ~My_Class() = default;
private:
  unique_ptr<int> p1;
  unique_ptr<int> p2;
};
```

Smart pointers

Better solution!

```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{create(y)}
{ }
```

Smart pointers

Better solution!

```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{create(y)}
{ }
```

Perfection!

www.liu.se

LINKÖPING
UNIVERSITY