TDDE18 & 726G77

Standard library

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

LINKÖPING UNIVERSITY

Containers

Introduction

- Sequence containers
- Sequence adaptors
- Associative containers

Containers

Sequence container

- `vector`
- `array`
- `deque`
- `list`
- `forward_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

```
#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

```
#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 };
}
```

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

LINKÖPING UNIVERSITY

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

```
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

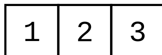
Iterators

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

Iterators

Iterators

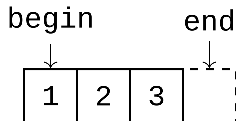
```
vector<int> v{1,2,3};
```



Iterators

Iterators

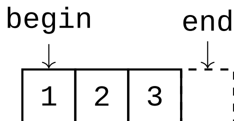
```
vector<int> v{1,2,3};
```



Iterators

Iterators

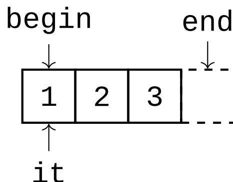
```
vector<int>::iterator it{v.begin()};
```



Iterators

Iterators

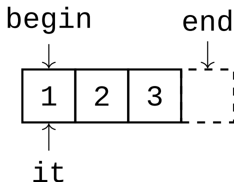
```
vector<int>::iterator it{v.begin()};
```



Iterators

Iterators

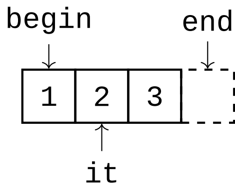
```
++it;
```



Iterators

Iterators

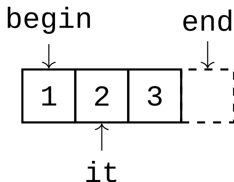
```
++it;
```



Iterators

Iterators

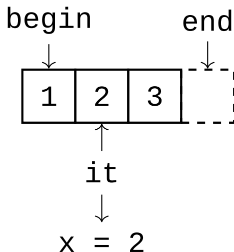
```
int x{*it};
```



Iterators

Iterators

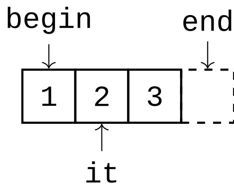
```
int x{*it};
```



Iterators

Iterators

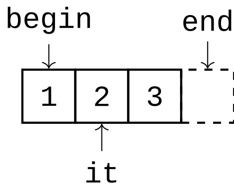
```
int x{*it};
```



Iterators

Iterators

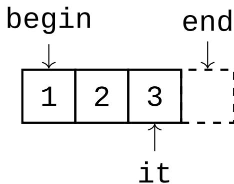
```
++it;
```



Iterators

Iterators

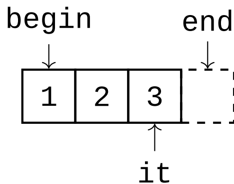
```
++it;
```



Iterators

Iterators

```
*it = 4;
```

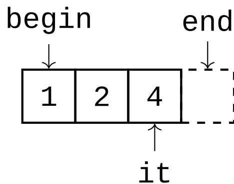


x = 2

Iterators

Iterators

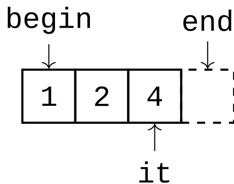
```
*it = 4;
```



Iterators

Iterators

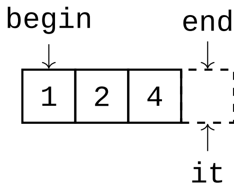
```
++it;
```



Iterators

Iterators

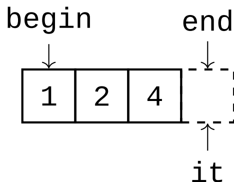
```
++it;
```



Iterators

Iterators

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



Iterators

Iterator categories

- Input
- Output
- Forward
- Bidirectional
- Random Access

Iterators

Iterator categories

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

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

Standard Library

What is the Standard Library?

- Available for everyone
- Solves common problems
- Components
- Effective

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

- Algorithms
- Containers
- Iterators
- Others

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

LINKÖPING UNIVERSITY

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?

```
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

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

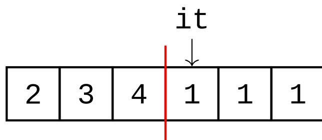
```
auto it {std::remove(v.begin(), v.end(), 1)};
```

1	2	3	1	4	1

Algorithms

Modifying algorithms

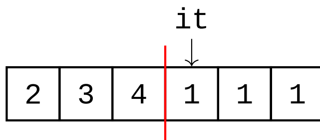
```
auto it {std::remove(v.begin(), v.end(), 1)};
```



Algorithms

Modifying algorithms

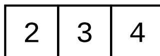
```
v.erase(it, v.end());
```



Algorithms

Modifying algorithms

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



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

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

```
// 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 categories

Some algorithms don’t work with all containers

Algorithms

Iterator categories

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

Algorithms

Iterator categories

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

Algorithms

Iterator categories

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

Doesn’t work!

Algorithms

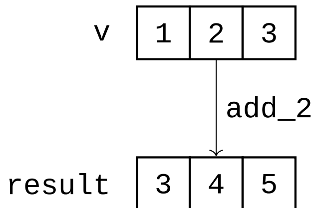
`std::transform`

```
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

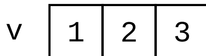
`std::transform`

```
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);
}
```

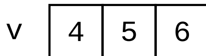
Algorithms

`std::transform`



Algorithms

`std::transform`



1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 **Lambda Functions**
6 More on iterators
7 Smart pointers

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

```
[](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

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

What is a lambda function?

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

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

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

```
[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

```
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

```
[&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

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}
```

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

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

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

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

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

```
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`

1 Containers
2 Iterators
3 Standard Library
4 Algorithms
5 Lambda Functions
6 More on iterators
7 Smart pointers

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

Solution?

```
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

Solution?

```
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

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?

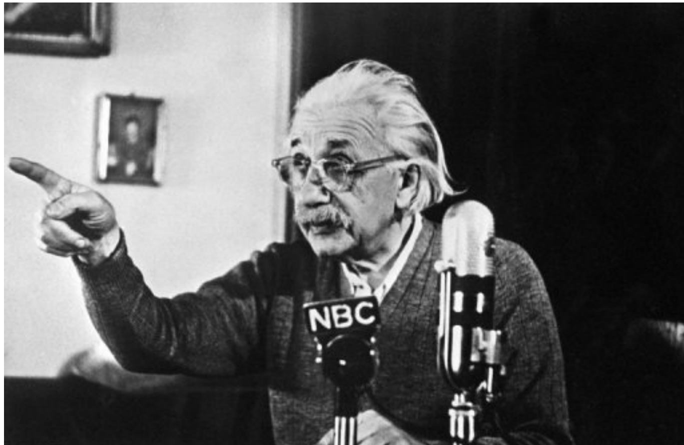
```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{}
{
  try
  {
    p2 = create(y);
  }
  catch (domain_error& e)
  {
    delete p1;
    throw;
  }
}
```

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

```
#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::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};
    }
  }
}
```

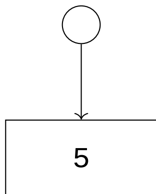
Smart pointers

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

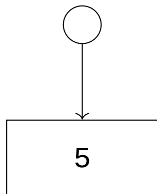
`std::shared_ptr`




```
 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

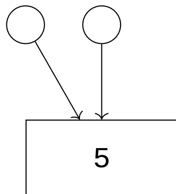
std::shared_ptr




```
  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

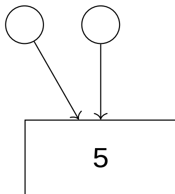
std::shared_ptr




```
 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

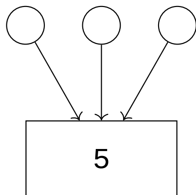
std::shared_ptr




```
 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

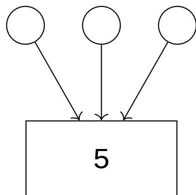
`std::shared_ptr`




```
 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

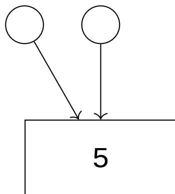
`std::shared_ptr`




```
 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

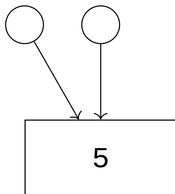
`std::shared_ptr`




```
 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

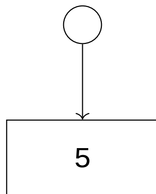
std::shared_ptr




```
 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

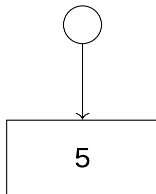
`std::shared_ptr`




```
 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

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