# TDP004 - Objektorienterd programmering

## Handout Standardbiblioteket

Anpassat från slides av Christoffer Holm

János Dani & Pontus Haglund

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

# Behållare

Introduktion

- Sekvensbehållare (Sequence containers)
- Sekvensadaptrar (Sequence adaptors)
- Associativa behållare (Associative containers)

# Behållare

Introduktion

- Sekvensbehållare (Sequence containers)
  - Lagrar värden av samma typ i en given sekvens
  - Vanligtvis används index för hämta värden
- Sekvensadaptrar (Sequence adaptors)
- Associativa behållare (Associative containers)

# Behållare

Introduktion

- Sekvensbehållare (Sequence containers)
- Sekvensadaptrar (Sequence adaptors)
  - Anpassat gränssnittet för en given sekvensbehållare
  - Representerar saker såsom stackar, köer, prioritets listor, o.s.v.
- Associativa behållare (Associative containers)

# Behållare

Introduktion

- Sekvensbehållare (Sequence containers)
- Sekvensadaptrar (Sequence adaptors)
- Associativa behållare (Associative containers)
  - Lagrar värden associerade till givna nycklar
  - Värden måste vara av samma datatyp
  - Nycklar måste vara av samma datatyp
  - Använder nycklarna för att komma åt värden

# Behållare

Sekvensbehållare

- `vector`
  - Lagrar värden kontinuerligt i minnet
  - Ändrar storlek efter behov
  - Vanligaste behållaren
- `array`
- `deque`

# Behållare

Sekvensbehållare

- `vector`
- `array`
  - Lagrar värden kontinuerligt i minnet
  - Har en fixerad storlek som är känd under kompilering
  - Är effektivare än `vector`
- `deque`

# Behållare

Sekvensbehållare

- `vector`
- `array`
- `deque`
  - **D**ouble-**e**nded **que**ue
  - Lagrar **inte** värden kontinuerligt i minnet
  - Bra om man vill komma åt värden endast i början OCH slutet

# Behållare

Sekvensbehållare

```
#include <vector>
#include <array>
#include <deque>

int main()
{
  // likadant för list, forward_list och deque
  std::vector<int>   v {1, 2, 3};

  // array måste även ange storlek
  std::array<int, 3> a {1, 2, 3};
}
```

# Behållare

Sekvensadaptrar

- `stack`

# Behållare

Sekvensadaptrar

- `stack`
  - Bygger ofta på `deque`
  - Kan endast komma åt det senaste inlagda värdet

# Behållare

Sekvensadaptrar

```
#include <stack>
#include <queue>
#include <priority_queue>
int main()
{
  // likadant för alla
  std::stack<int> s1 {};

  // kan ändra vilken behållare
  std::stack<int, std::vector<int>> s2 {};
}
```

# Behållare

Associativa behållare

- `map`
- `set`

# Behållare

Associativa behållare

- `map`
  - Kopplar ett värde till en nyckel
  - Kräver att nycklarna går att jämföra
  - Sorterad efter nycklarna
  - Varje nyckel kan endast förekomma en gång
- `set`

# Behållare

Associativa behållare

- `map`
- `set`
  - Som `map` men har endast nycklar
  - Bra för att garantera att alla värden är unika och sorterade

# Behållare

Associativa behållare

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

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

# Iteratorer

Iteration

- Vi vill kunna loopa igenom våra behållare
- Vore trevligt om vi kan göra det generellt
- Problemet är att alla behållare har inte samma sätt att komma åt element
- Därför måste vi tänka om för att kunna loopa igenom behållare på ett generellt sätt

# Iteratorer

Iteration

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

# Iteratorer

Iteration

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

# Iteratorer

Iteration

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

# Iteratorer

Iteration

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

# Iteratorer

Range-baserad for-loop

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

# Iteratorer

Range-baserad for-loop

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

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

# Iteratorer

Iteratorer

- Iteratorer är generaliserade pekare
- Ett generellt sätt att iterera över alla behållare på samma sätt
- Pekar på ett element i behållaren
- Möjligt att komma åt elementet med `operator*`
- Gå till nästa element med `operator++`

# Iteratorer

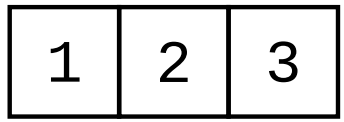
Iteratorer

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

# Iteratorer

Iteratorer

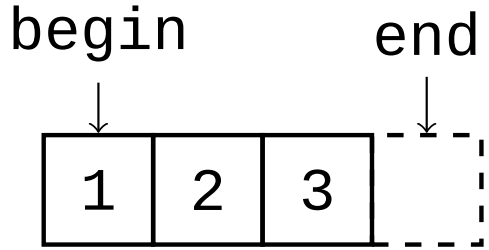
```
vector<int> v{1,2,3};
```



# Iteratorer

Iteratorer

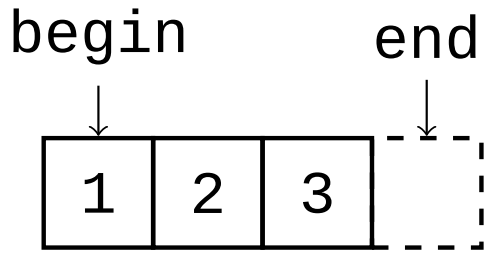
```
vector<int> v{1,2,3};
```



# Iteratorer

Iteratorer

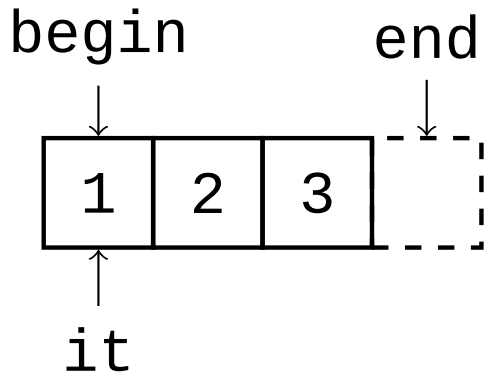
```
vector<int>::iterator it{v.begin()};
```



# Iteratorer

Iteratorer

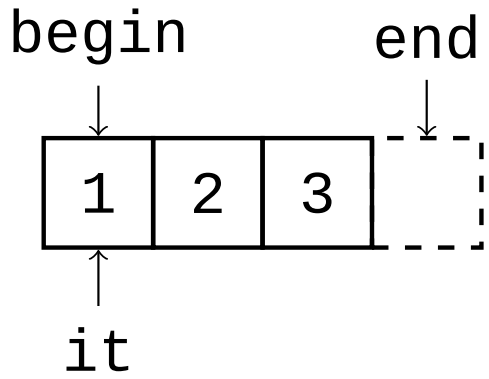
```
vector<int>::iterator it{v.begin()};
```



# Iteratorer

Iteratorer

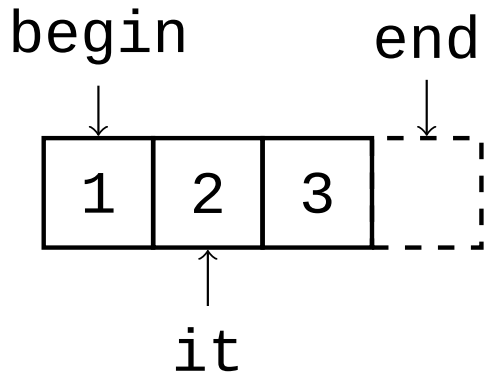
```
++it;
```



# Iteratorer

Iteratorer

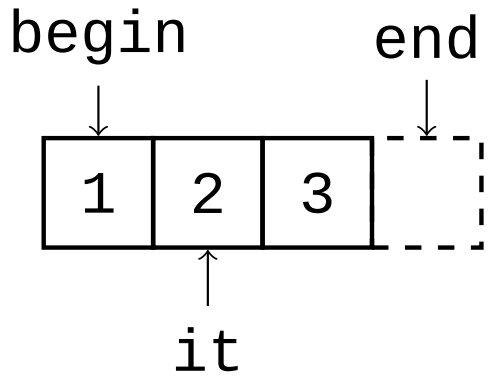
```
++it;
```



# Iteratorer

Iteratorer

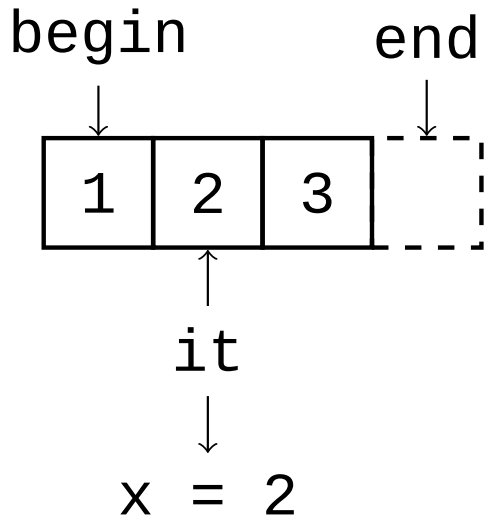
```
int x{*it};
```



# Iteratorer

Iteratorer

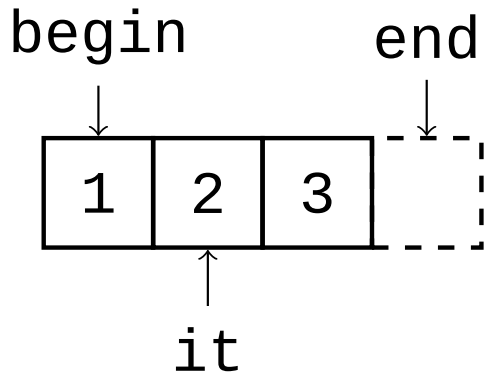
```
int x{*it};
```



# Iteratorer

Iteratorer

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

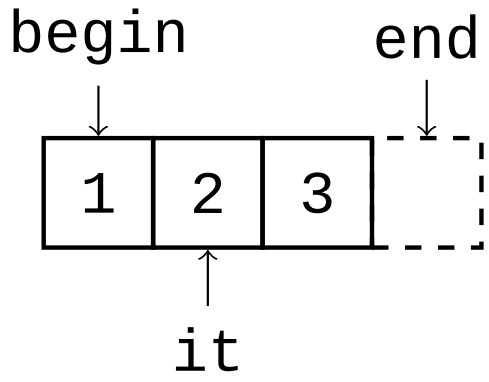



x = 2

# Iteratorer

Iteratorer

```
++it;
```

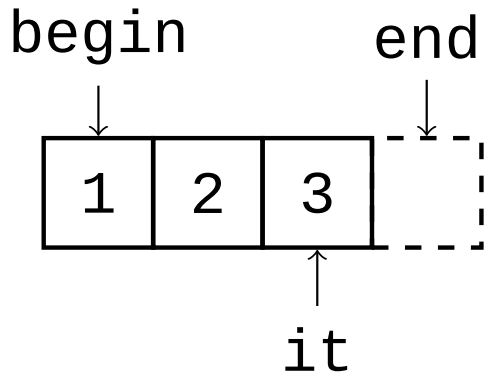



x = 2

# Iteratorer

Iteratorer

```
++it;
```

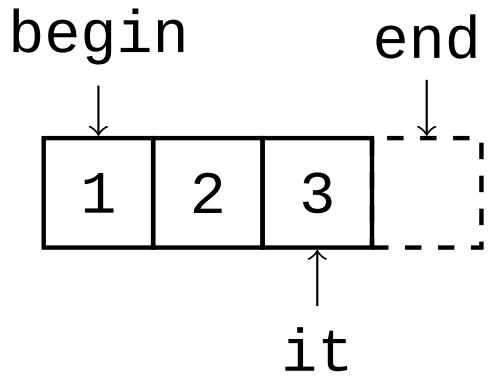



x = 2

# Iteratorer

Iteratorer

```
*it = 4;
```

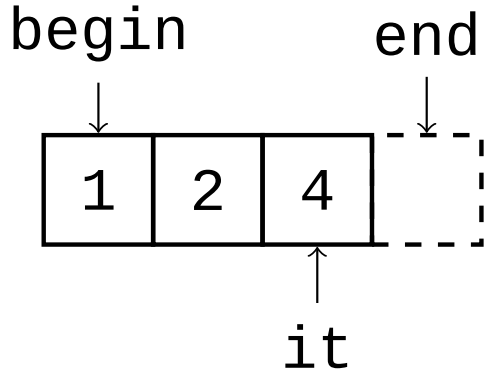



x = 2

# Iteratorer

Iteratorer

```
*it = 4;
```

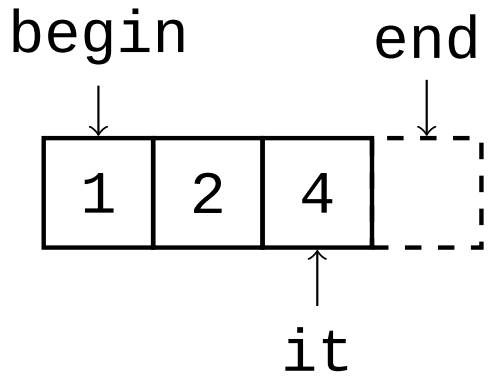



x = 2

# Iteratorer

Iteratorer

```
++it;
```

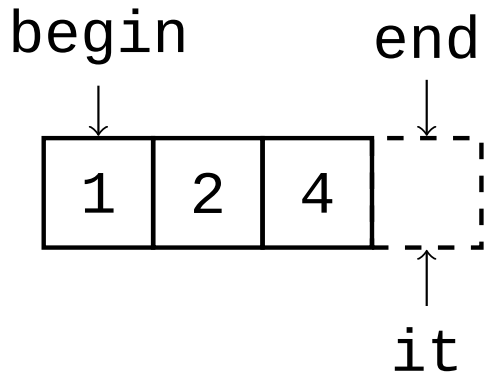



x = 2

# Iteratorer

Iteratorer

```
++it;
```

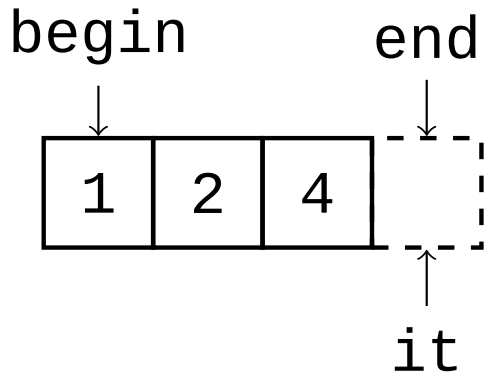



x = 2

# Iteratorer

Iteratorer

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




x = 2

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

# Smartpekare

Kan det bli minnesfel här?

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

# Smartpekare

Kan det bli minnesfel här?

```
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)}
{ }
```

# Smartpekare

Ja, det kan bli fel!

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

# Smartpekare

Varför?

- När konstruktorn avbryts kommer objektet att tas bort utan att köra destruktorn
- All data som har allokerats innan krashen kommer därför inte avallokeras
- Hur kan vi lösa detta?

# Smartpekare

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};
}
```

# Smartpekare

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};
}
```

Segmentation Fault

# Smartpekare

Varför?

- Nu är det `p1` som kastar undantaget
- I catch-blocket försöker vi ta bort den, men den finns inte
- Detta ger segmentation fault

# Smartpekare

Lösning?

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

# Smartpekare

Lösning?

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

Funkar...

# Smartpekare

Lösning?

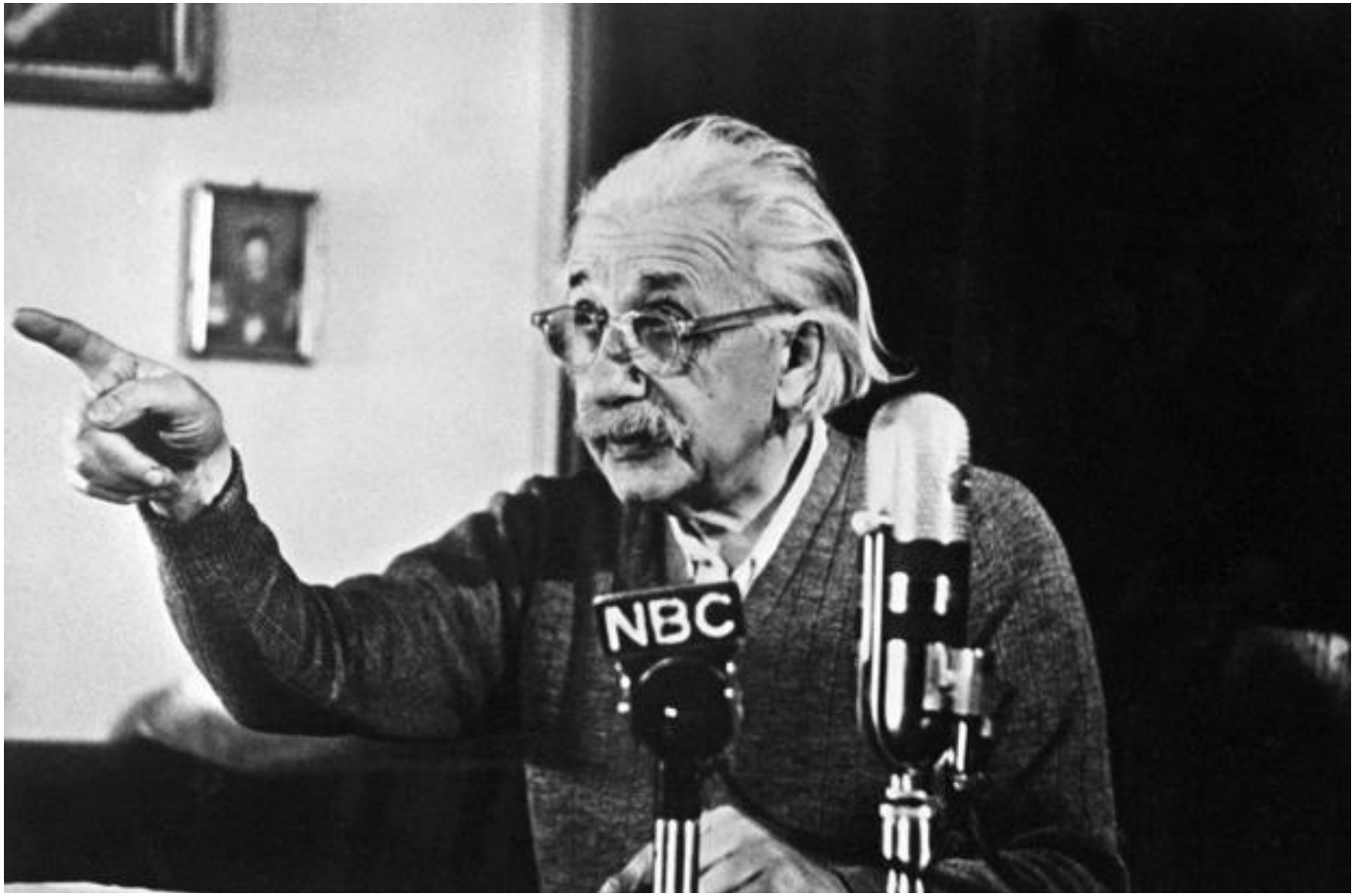
```
My_Class::My_Class(int x, int y)
  : p1{create(x)}, p2{}
{
  try
  {
    p2 = create(y);
  }
  catch (domain_error& e)
  {
    delete p1;
    throw;
  }
}
```

Men till vilket pris?

## Smartpekare

# Vore det inte bra om pekare kunde avallokera sig själva?

# Smartpekare



# Smartpekare

`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;
}
```

# Smartpekare

`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;

  }

}
```

# Smartpekare

`std::unique_ptr`

- är en s.k. *smartpekare*
- finns definierad i `<memory>`
- tar över ansvaret för att hantera minnet
- representerar *ägarskap*
- kan **inte** kopieras
- men det går att flytta ägarskapet

# Smartpekare

`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);
  }
}
```

# Smartpekare

`std::unique_ptr`

- `unique_ptr` tar bort det gamla minnet när vi tilldelar nytt minne
- man ska nästan *aldrig* behöva avallokera minnet manuellt

# Smartpekare

Fälla

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

# Smartpekare

Fälla

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

Kompilerar ej

# Smartpekare

Fälla

```
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)
     ^~~
```

# Smartpekare

Fälla

```
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)
     ^~~
```

# Smartpekare

Fälla

Om du ser detta så försöker du kopiera en `unique_ptr`

# Smartpekare

Mer om `unique_ptr`

```
int main()
{
  std::unique_ptr<std::string> p{};
  // abstrahera bort allokeringen
  p = std::make_unique<std::string>("hej");
  // plocka ut en vanlig pekare
  std::string* ptr{p.get()};
  // kom åt medlemmar i det som pekaren pekar på
  cout << p->size() << endl;
}
```

# Smartpekare

`std::make_unique`

- Om vi använder `std::make_unique` snarare än `new` kommunicerar vi bättre vad som händer i koden
- Det blir tydligt att koden inte kräver en `delete` om `new` inte ens förekommer
- Detta tillåter kompilatorn är resonera bättre kring koden och kan därför göra eventuella optimeringar som inte annars går

# Smartpekare

`get`

- `std::unique_ptr::get` ska endast användas då vi behöver temporär tillgång till objektet
- den pekare som returneras av `get` är en *icke-ägande* pekare
- detta innebär att du aldrig någonsin ska göra `delete` på den pekaren
- om du av någon annledning vill avallokera minnet, tilldela `nullptr` till smartpekaren eller anropa `release` funktionen

# Smartpekare

`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};
    }
  }
}
```

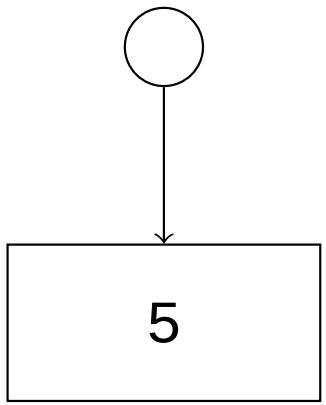
# Smartpekare

`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};
     }
   }
 }
```

# Smartpekare

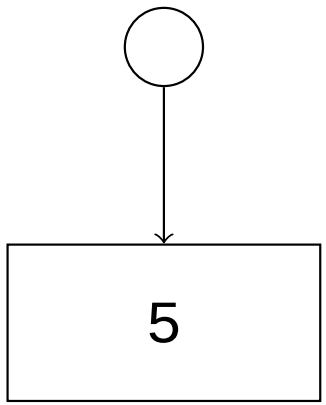
`std::shared_ptr`




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

# Smartpekare

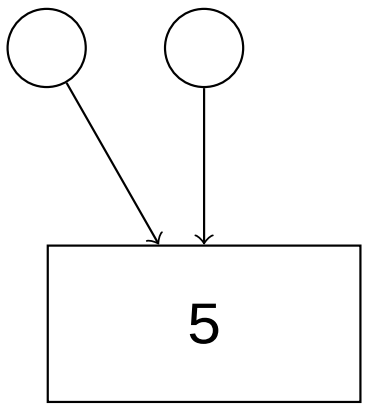
`std::shared_ptr`




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

# Smartpekare

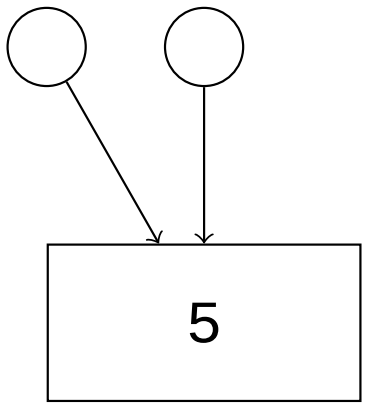
`std::shared_ptr`




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

# Smartpekare

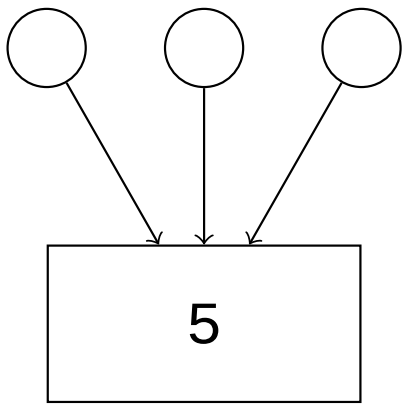
`std::shared_ptr`




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

# Smartpekare

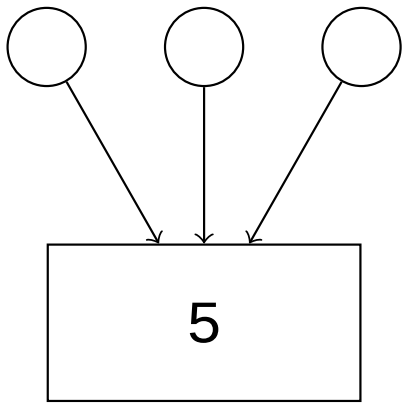
`std::shared_ptr`




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

# Smartpekare

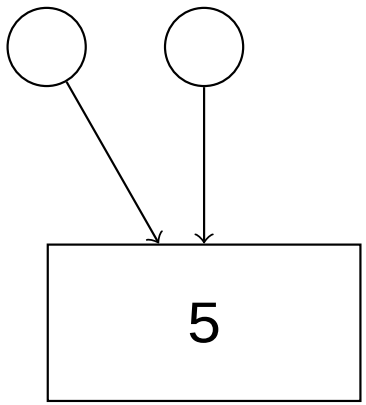
`std::shared_ptr`




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

# Smartpekare

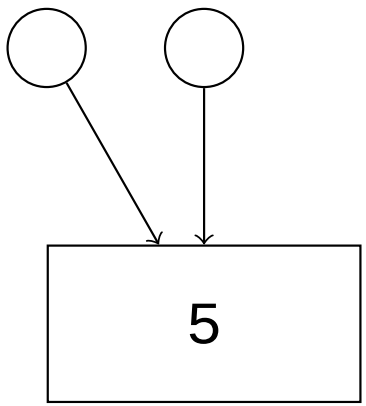
`std::shared_ptr`




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

# Smartpekare

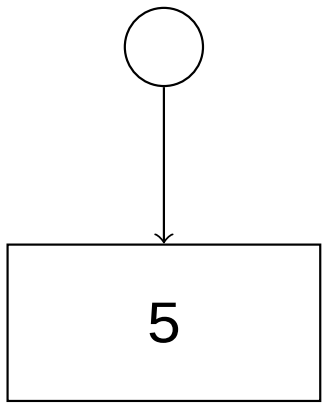
`std::shared_ptr`




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

# Smartpekare

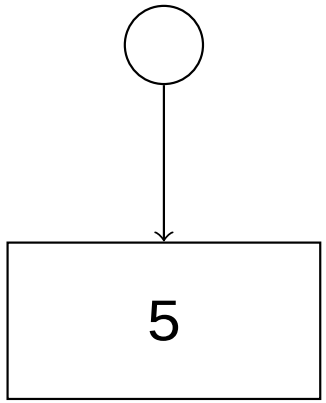
`std::shared_ptr`




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

# Smartpekare

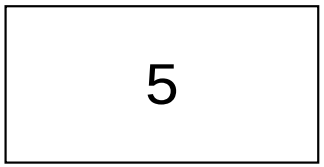
`std::shared_ptr`




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

# Smartpekare

`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};
     }
   }
>}
```

# Smartpekare

`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};
     }
   }
>}
```

# Smartpekare

`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};
    }
  }
}
```

# Smartpekare

`std::shared_ptr`

- Representerar delat ägarskap
- Minnet avallokeras endast då ingen längre pekar på det
- Kostar mer än `std::unique_ptr` och vanliga pekare

# Smartpekare

Hur hjälper detta oss?

# Smartpekare

Bättre lösning!

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

# Smartpekare

Bättre lösning!

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

# Smartpekare

Bättre lösning!

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

Perfektion!

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

# Standardbiblioteket

Vad är standardbiblioteket?

- Tillgängligt för alla
  - Samma beteende oberoende av dator och operativ system
  - Följer med kompilatorn
  - ISO C++ kräver att allt är implementerat
- Löser vanliga problem
- Samling komponenter
- Effektivt

# Standardbiblioteket

Vad är standardbiblioteket?

- Tillgängligt för alla
- Löser vanliga problem
  - Att uppfinna hjulet tar tid
  - Det finns problem som alla programmerare stöter på
  - Ska vara brett användbart
- Samling komponenter
- Effektivt

# Standardbiblioteket

Vad är standardbiblioteket?

- Tillgängligt för alla
- Löser vanliga problem
- Samling komponenter
  - Betala inte för vad du inte använder
  - Importera endast delarna du behöver
  - Allt är kompatibelt med varandra
- Effektivt

# Standardbiblioteket

Vad är standardbiblioteket?

- Tillgängligt för alla
- Löser vanliga problem
- Samling komponenter
- Effektivt
  - De som skriver biblioteket kan sin sak
  - Allting är väldigt optimerat
  - Det är inte ditt ansvar att se till att allt fungerar

# Standardbiblioteket

STL

**S**tandard **T**emplate **L**ibrary

# Standardbiblioteket

Designmål

- Ska vara så generellt som möjligt
- Löser vanliga problem
- Det vanliga fallet ska vara enkelt
- Måste fungera med användarens kod
- Ska vara effektiv nog för att ersätta handskrivna alternativ
- Ska ha robust felhantering

# Standardbiblioteket

Komponenter

- Algoritmer
  - Generella funktioner för att lösa vanliga problem
  - Utför operationer på databehållare
  - Använder iteratorer som gränsnittet mot behållare
  - Optimerat för hastighet och minne
- Databehållare
- Iteratorer
- Övrigt

# Standardbiblioteket

Komponenter

- Algoritmer
- Databehållare
  - Olika sätt att strukturera data
  - Baseras på abstraktioner
  - Vi ska inte behöva veta hur de fungerar
- Iteratorer
- Övrigt

# Standardbiblioteket

Komponenter

- Algoritmer
- Databehållare
- Iteratorer
  - Gränssnitt för att traversera data
  - Används för att abstrahera bort behållare
- Övrigt

# Standardbiblioteket

Komponenter

- Algoritmer
- Databehållare
- Iteratorer
- Övrigt
  - Generella funktioner och klasser
  - Löser diverse vanliga problem
  - Ska kunna användas för så många typer som möjligt

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

# Algoritmer

Varför?

- Standard algoritmer tillåter oss att tydligare kommunicera vad koden gör
- Andra programmerare förstår standard algoritmerna väldigt snabbt, medan de kräver mer tid för dem att sätta sig in i vad handskrivna alternativ gör
- Det blir mindre steg för oss att tänka på

# Algoritmer

Vad gör denna kod?

```
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())
{
  // hittade inget
}
```

# Algoritmer

Vad gör denna kod?

```
std::vector<int> v { 5, -2, 8, 4, 7 };
auto it {std::find(v.begin(), v.end(), 4)};
if (it == v.end())
{
  // hittade inget
}
```

# Algoritmer

Vad gör denna kod?

Vilken variant är lättast att förstå?
I den andra varianten används en algoritm som bokstavligen heter **find** för att hitta ett element i listan.

# Algoritmer

Vad gör denna kod?

- Algoritmer gör koden mer lättläslig,
- slipper skriva samma kod om och om igen,
- kan tänka på en högre nivå,
- behöver inte tänka (lika mycket) på optimalitet

# Algoritmer

Vad gör denna kod?

```
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++;
}
```

# Algoritmer

Vad gör denna kod?

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

# Algoritmer

Vilka algoritmer finns det?

- Det finns över 100 olika algoritmer tillgängliga
- För varje version av C++ tillkommer det nya
- En komplett lista finns här:
  `https://en.cppreference.com/w/cpp/algorithm`

# Algoritmer

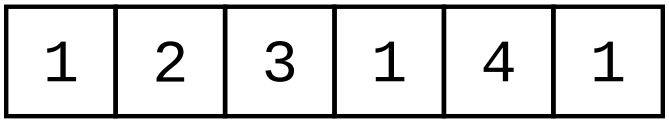
Modifierande algoritmer

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

# Algoritmer

Modifierande algoritmer

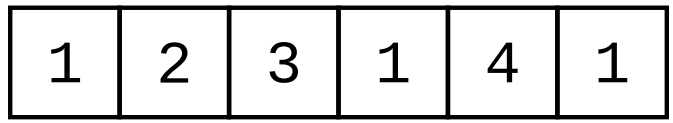
```
std::vector<int> v {1, 2, 3, 1, 4, 1};
```



# Algoritmer

Modifierande algoritmer

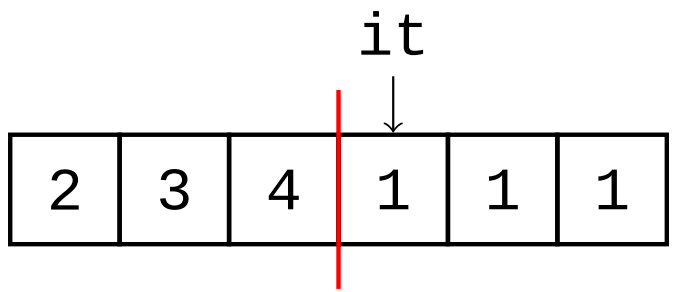
```
auto it {std::remove(v.begin(), v.end(), 1)};
```



# Algoritmer

Modifierande algoritmer

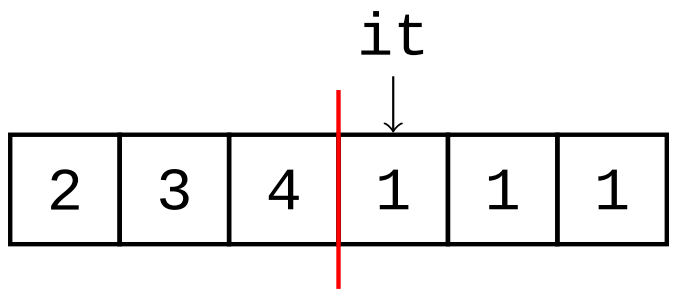
```
auto it {std::remove(v.begin(), v.end(), 1)};
```



# Algoritmer

Modifierande algoritmer

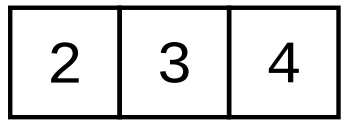
```
v.erase(it, v.end());
```



# Algoritmer

Modifierande algoritmer

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



# Algoritmer

Modifierande algoritmer

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

# Algoritmer

Modifierande algoritmer

- Vissa algoritmer *tar bort* element
- Det finns ingen funktionalitet i iteratorer som tillåter oss att faktiskt ta bort (eller lägga till) element
- Det som händer istället är att den flyttar alla *borttagna* element till slutet av behållaren och returnerar en iterator till det första av dessa element
- Därefter får användaren avgöra hur den faktiska borttagningen ska tas bort (oftast med `erase`)

# Algoritmer

Kopiering

```
// kommandoradsargument
std::vector<string> args {argv, argv + argc};

// tom vektor
std::vector<string> v {};

// kopiera alla argument till den tomma vektorn
std::copy(args.begin(), args.end(), v.begin());
```

# Algoritmer

Kopiering

- Den tomma vektorn har inga element
- Dessa iteratorer kan endast läsa och skriva över befintliga element
- Därför försöker vi kopiera element från `args` till en vektor (v) som inte har några platser att skriva till

# Algoritmer

Kopiering

```
// kommandoradsargument
std::vector<string> args {argv, argv + argc};

// vektor med rätt antal platser
std::vector<string> v (args.size());

// kopiera alla argument till den tomma vektorn
std::copy(args.begin(), args.end(), v.begin());
```

# Algoritmer

Iterator kategori

Vissa algoritmer funkar inte för alla databehållare

# Algoritmer

Iterator kategori

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

# Algoritmer

Iterator kategori

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

# Algoritmer

Iterator kategori



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

# Algoritmer

Iterator kategori

- Varför funkar det inte för `std::list`?
- Jo, för att kunna sortera data måste vi kunna hoppa mellan godtyckliga element i databehållaren
- Därför måste vi ha en `RandomAccessIterator` (den som har `operator+` o.s.v.)
- `std::list` har endast `BidirectionalIterator`

# Algoritmer

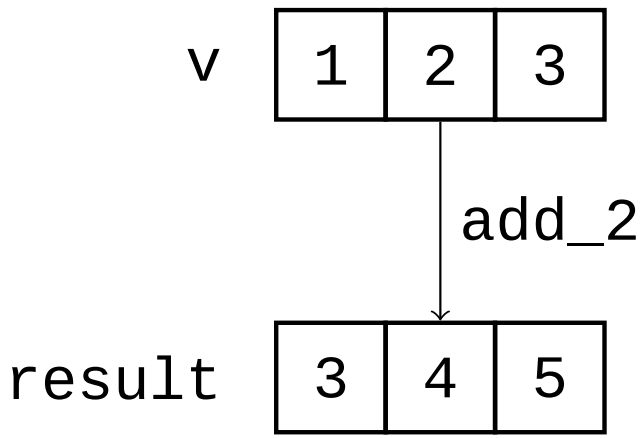
`std::transform`

```
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);
}
```

# Algoritmer

`std::transform`



# Algoritmer

`std::transform`

- `std::transform` fungerar som `std::copy`...
- men den tillämpar först den angivna funktionen på elementen
- i detta fall innebär det att vi kopierar varje element från `v`, men lägger först till `2` till värdet innan det hamnar i `result`

# Algoritmer

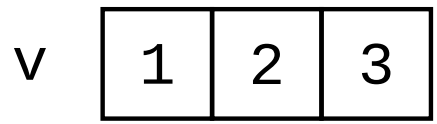
`std::transform`

```
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);
}
```

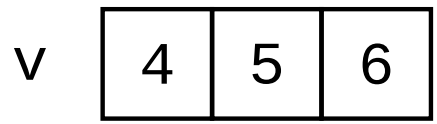
# Algoritmer

`std::transform`



# Algoritmer

`std::transform`



# Algoritmer

`std::transform`

- Det går även bra att använda `std::transform` för att skriva till samma vektor som vi läser ifrån
- Väldigt vanligt att man gör så
- Kräver självklart att returtypen från funktionen är densamma som värdena i vektorn

# Algoritmer

`std::transform`

- I många fall vill tillämpa en operation på varje element endast en gång i hela programmet
- Det kan väldigt snabbt bli många funktioner deklarerade
- funktioner som endast används en gång
- Det vore trevligt om vi kan skapa temporära funktioner som kan skapas i samband med vårt `std::transform` anrop...

LINKÖPING UNIVERSITY

# Lambdafunktioner

Tillfälliga funktioner

- En lambdafunktion är ett uttryck som skapar en temporär funktion
- Tillåter oss att skapa och använda funktioner utan att ge dem ett namn
- Är även mer generella än vanliga funktioner

# Lambdafunktioner

Tillfälliga funktioner

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

# Lambdafunktioner

Tillfälliga funktioner

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

| | |
|---|---|
| `[]` | capture |
| `(int n)` | inparameter |
| `-> int` | returtyp |
| `{ return n + 2; }` | kropp |

# Lambdafunktioner

Returtyp

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

# Lambdafunktioner

Returtyp

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

# Lambdafunktioner

Returtyp

- Vi kan ha `auto` som returtyp
- Men det härliga med lambdafunktioner är att detta är automatiskt vad som händer om vi inte specificerar returtyp överhuvudtaget

# Lambdafunktioner

Returtyp

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

# Lambdafunktioner

`std::transform`

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

# Lambdafunktioner

Lambdafunktioner och STL

- Det finns många algoritmer som tar funktioner som argument
- Oerhört vanligt att vi använder lambdafunktioner i dessa sammanhang
- Koden blir kortare och enklare att läsa
- Användaren behöver inte hitta funktionsdeklarationen och kan direkt se vad lambdafunktionen gör

# Lambdafunktioner

Namnge lambdafunktioner

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

# Lambdafunktioner

Namnge lambdafunktioner

- På detta sätt kan vi behålla abstraktionen som uppstår med att anropa funktioner med namn
- men vi behöver inte tvinga användaren att leta efter funktionens definition långt bort, utan den finns tillgänglig precis ovanför
- Men betyder inte detta att `add_2` nu är en variabel?
- Jo det gör det!

# Lambdafunktioner

Vad är en lambdafunktion? ÖVERKURS

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

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

# Lambdafunktioner

`operator()` ÖVERKURS

- `operator()` kallas *funktionsanropsoperatorn*
- Om jag har ett objekt `obj` som är av en datatyp som har definierat `operator()` så översätts: `obj(x)` till `obj.operator()(x)`
- Alla klasser med `operator()` kallas *funktionsobjekt*

# Lambdafunktioner

Vad är en lambdafunktion? ÖVERKURS

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

```
My_Lambda add_2 {};
```

# Lambdafunktioner

Capture

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

# Lambdafunktioner

Capture

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

# Lambdafunktioner

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; });
```

# Lambdafunktioner

Capture

- Innanför `[]` kan man skriva in vilka variabler som ska finnas tillgängliga inuti lambdafunktionen
- Detta kallas för lambdafunktionens *capture*
- Detta kommer skapa en lokal kopia av variablerna

# Lambdafunktioner

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;
};
```

# Lambdafunktioner

Capture

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

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

# Lambdafunktioner

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
```

# Lambdafunktioner

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
```

# Lambdafunktioner

Capture

- Om vi lägger ett & framför variabelns namn i capture kommer den bindas som en referens istället

# Lambdafunktioner

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;
};
```

# Lambdafunktioner

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}
```

# Lambdafunktioner

Capture all

- Om vi endast skriver `[&]` innebär det att vi fångar alla variabler som finns tillgängliga vid lambdafunktionens definitionstillfället
- Fångar egentligen bara de variabler som används inuti lambdafunktionen
- Fångar allt som en referens

# Lambdafunktioner

Funktionsobjekt i STL

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

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

# Lambdafunktioner

Funktionsobjekt i 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}
```

# Lambdafunktioner

Funktionsobjekt i STL

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

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

# Lambdafunktioner

Funktionsobjekt i STL

- Det finns en del inbyggda funktionsobjekt som kan användas med algoritmer
- Användbara exempel: `std::less`, `std::greater`, `std::plus` m.fl.
- De alla finns listade här: `https://en.cppreference.com/w/cpp/utility/functional` (**Operator function objects**)

1 Behållare
2 Iteratorer
3 Smartpekare
4 Standardbiblioteket
5 Algoritmer
6 Lambdafunktioner
7 Mer om iteratorer

LINKÖPING UNIVERSITY

# Mer om iteratorer

`std::for_each` UNDVIK DENNA ALGORITM

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

# Mer om iteratorer

`std::for_each` UNDVIK DENNA ALGORITM

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

# Mer om iteratorer

`std::for_each`

- `std::for_each` är en kvarleva från en äldre dagar
- C++ har utvecklats till den nivån att `std::for_each` *nästan aldrig* ska behövas längre
- undvik `std::for_each` så ofta det går

# Mer om iteratorer

Skriva ut till en behållare

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

# Mer om iteratorer

Skriva ut till en behållare

Skriver ut:

```
1234
```

# Mer om iteratorer

Skriva ut till en behållare

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

# Mer om iteratorer

Skriva ut till en behållare

Skriver ut:

```
1 2 3 4
```

# Mer om iteratorer

`std::ostream_iterator`

- Är en `OutputIterator`
- Givet `std::ostream_iterator<int> it {cout}` kommer uttrycket `*it = 5` att skriva ut `5` till `cout`
- `++it` och `it++` gör ingenting

# Mer om iteratorer

Läsa från en ström

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

# Mer om iteratorer

Läsa från en ström

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

# Mer om iteratorer

Läsa från en ström

- `std::vector` har en konstruktor som kopierar värden från ett par iteratorer
- givet en start och slut-iterator kommer denna konstruktor att kopiera varje värde och stoppa in dessa i vektorn

# Mer om iteratorer

Läsa från en ström

- `std::istream_iterator` är en *InputIterator*
- Givet `std::istream_iterator<int> it cin` så kommer `*it` ta ut det senaste lästa värdet från `cin`
- `it++` eller `++it` kommer läsa in nästa värde från `cin`
- Kan används som om `cin` är en behållare
- Default-konstruktorn skapar *slut*-iteratorn

# Mer om iteratorer

Output iteratorer

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

# Mer om iteratorer

Output iteratorer

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

# Mer om iteratorer

Output iteratorer

- `back_inserter` skapar en *OutputIterator* som när vi tilldelar den så anropar den `push_back` på den underliggande behållaren
- Givet `std::vector<int> v` och `auto it{std::back_inserter(v)}` kommer `*it++ = 5` vara ekvivalent med att anropa `v.push_back(5)`
- Mycket användbart tillsammans med algoritmer såsom `std::copy` och `std::transform`

# Mer om iteratorer

```
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, ", "});
```

# Mer om iteratorer

Om vi nu kör:

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

# Mer om iteratorer

Skriver ut:

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

# Tips inför Ordlistelabben:

kolla på alla medlemsfunktioner i
`std::string`

www.liu.se

LINKÖPING UNIVERSITY