

TDP004

STL

Eric Ekström

Institutionen för datavetenskap

Handout version

- 1 Intro
- 2 Behållare
- 3 Iteratorer
- 4 Algoritmer
- 5 Lambda-uttryck

Standardbiblioteket

Standard Template Library

Standardbiblioteket

Designprinciper:

- Ska vara så generellt som möjligt
- Löser vanliga problem
- Måste fungera med användarens kod
- Effektivt nog att ersätta egengjorda lösningar
- Robust felhantering

- 1 Intro
- 2 Behållare**
- 3 Iteratorer
- 4 Algoritmer
- 5 Lambda-uttryck

STL - std::vector

```
#include <vector>

int main()
{
    std::vector<int> v {};
    v.push_back(1);
    v.push_back(5);
    v.push_back(2);

    std::cout << v.at(0) << v.at(1) << v.at(2) << std::endl;
    return 0;
}
```

STL - std::set

```
#include <set>

int main()
{
    std::set<double> s { 1.0, 3.0, -1.0 };

    s.insert(3.14);
    s.erase(1.0);
}
```

En sorterad mängd av värden.

- Snabb insättning och åtkomst
- Ingen indexering
- Tillåter inte dubletter.

STL - std::deque

```
#include <deque>

int main()
{
    std::deque<char> d { 'h', 'e', 'j' };

    d.push_back('!');
    d.pop_front();
}
```

- Generellt lite långsammare än std::vector, men
- Insättning i början och slutet snabbt

STL - std::map

```
std::map<std::string, int> m { {"a", 1},  
                               {"b", 2} };  
  
// Åtkomst av befintligt värde  
std::cout << m["a"];  
m["b"] = 7;  
  
// Sätter in ett nytt värde  
m["c"] = 2;
```

- Associativ behållare.
- Sparar ett par av [nyckel, värde]
- Nycklar måste gå att jämföra

- 1 Intro
- 2 Behållare
- 3 Iteratorer**
- 4 Algoritmer
- 5 Lambda-uttryck

Iteratorer

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

Iteratorer

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

Iteratorer

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

Iteratorer

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

Iteratorer

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

Iteratorer

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

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

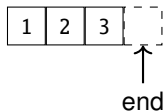
Iteratorer

Iteratorer är generaliserade pekare.

- `begin()` och `end()` ger iteratorer till början respektive slutet av en behållare.
- En iterator har (som minst) funktionerna
 - `operator++`
 - `operator*`
 - `operator==`
 - `operator!=`
- Olika implementation beroende på behållare

Iteratorer

Vart pekar `end()`?



Varför?

- Vissa behållare har inte en tydlig bild av sista elementet.
- Vad händer om behållaren är tom?
- Det underlättar iteration.

- 1 Intro
- 2 Behållare
- 3 Iteratorer
- 4 Algoritmer**
- 5 Lambda-uttryck

Algoritmer

Vad gör denna kod?

```
int main()
{
    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?

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

Vilka algoritmer finns? www.cppreference.com!

Algoritmerna ger oss ett mer lättläst sätt att utföra vanliga operationer. Standardbiblioteket ger oss ett enhetligt gränssnitt som alla algoritmer följer.

Målet är inte att memorera alla algoritmer som finns i C++ . Fokusera istället på att förstå hur algoritmerna är uppbyggda, vilka de gemensamma mönstren är och hur du går tillväga för att hitta en algoritm.

Algoritmer

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

Vilka mönster ser vi?

- Alla algoritmer tar två iteratorer
- Ger ut ett resultat
 - `std::count` -> `int`
 - `std::find` -> `std::vector<int>::iterator`

Att alltid ta två iteratorer ger oss flexibilitet. Tex om vi endast vill söka genom en del av behållaren.

Att ge ut en iterator till ett objekt är mer generellt. Funderar även om inget värde hittades (returnera end-iteratorn!)

Modifierande algoritmer

Om vi vill ändra i en behållare då?

```
std::vector<int> v {1, 2, 3, 1, 4, 1};  
std::fill(v.begin(), v.end(), 8);
```

`std::fill` sätter varje element i behållaren till det angivna värdet.

Modifierande algoritmer

Hur gör vi om vi vill ta bort värden?

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

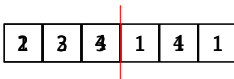
Fungerar inte!

Vad som händer om vi tar bort ett element medans vi itererar beror på behållaren.

- `std::vector` - ett element skippas
- `std::list` - iteratorn pekar nu på något odefinierat

Modifierande algoritmer

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



`std::remove` flyttar alla element som matchar till slutet av behållaren och ger tillbaka en pekare till första borttagna elementet.
`std::erase` tar bort alla element mellan två iteratorer. I detta fall början av de borttagna elementen och slutet av listan.

Modifierande algoritmer

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

Inläsning med algoritmer

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

```
using namespace std;  
  
int main()  
{  
    vector<int> v {};  
    copy(istream_iterator<int>{cin},  
         istream_iterator<int>{},  
         back_inserter(v));  
}
```

Utskrift med algoritmer

```
for (int x : v)
{
    cout << x << " ";
}
```

```
using namespace std;

int main()
{
    vector<int> v {};
    copy(v.begin(), v.end(),
         ostream_iterator<int>{cout, " "});
}
```

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);
}
```

Måste vi skriva funktionen separat?

- 1 Intro
- 2 Behållare
- 3 Iteratorer
- 4 Algoritmer
- 5 Lambda-uttryck

Lambda

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

Lambda - capture

```
int main()
{
    int n {};
    std::cin >> n;
    std::vector<int> w(v.size());

    if (n == 1)
    {
        std::transform(v.begin(), v.end(), w.begin(),
                       [](int x) {return x + 1;});
    }
    else if (n == 2)
    {
        std::transform(v.begin(), v.end(), w.begin(),
                       [](int x) {return x + 2;});
    }
    //...
}
```

Dåligt!

Lambda - capture

```
std::transform(v.begin(), v.end(), w.begin(),
               [n](int x)
               {
                   return x + n;
               });
```

Lambda - capture

```
int n { 2 };  
std::vector<int> w(v.size());  
  
auto add = [&n](int x) { return x + n; };  
  
std::transform(v.begin(), v.end(), w.begin(), add);  
n = 3;  
std::transform(v.begin(), v.end(), w.begin(), add);  
n = 5;  
std::transform(v.begin(), v.end(), w.begin(), add);
```

Lambda-uttryck

```
[capture](parametrar) -> returtyp  
{  
    funktionsblock  
}
```

www.ida.liu.se/~TDP004/