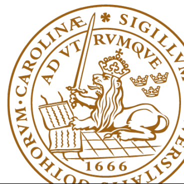
EDAF30 – Programmering i C++

Standardbiblioteket: Datastrukturer och algoritmer

Sven Gestegård Robertz
Datavetenskap, LTH

2015



Innehåll

1. Datastrukturer, containerklasser
   - Sekvenser
   - Iteratorer
   - Insättning
2. Algoritmer
   - Insättningsiteratorer
   - Funktionsobjekt
3. Mer datastrukturer
   - Listor
   - Avbildningar och mängder
   - Adapterklasser: Köer och stackar

Containerklasser – Klassificering

Sekvenser
- vector<T>
- deque<T>
- list<T>

adapterklasser (begränsade versioner av ovanstående)
- queue<T, Sequence>
- priority_queue<T, Sequence>
- stack<T, Sequence>

Associativa containers (finns även *unordered*)
- map<K,V>, multimap<K,V>
- set<T>, multiset<T>

Klasserna vector och deque

Med *containerklasser* avses klasser som beskriver datasamlingar av olika slag, t.ex. vektorer, listor, mängder eller avbildningar

Två typiska exempel
- Klassen vector
- Klassen deque(double-ended queue)

Kräver direktiven (respektive)

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


Klasserna vector och deque

Operationer i klassen vector

```
v.clear(), v.size(), v.empty()
v.push_back(), v.pop_back()
v.front(), v.back(), v.at(i), v[i]
v.assign(), v.insert(), v.emplace()
v.resize(), v.reserve()
```

Ytterligare operationer i klassen deque

```
d.push_front(), d.pop_front()
```


Klasserna vector och deque

Typen av element anges som *template-parameter* (inom <>)

Default-konstruktor

```
vector<double> vd1;     // vektor med flyttal (0 element)
vector<int> vi1;        // vektor med heltal (0 element)
```

Konstruktorer: (storlek), (storlek,elementvärde) och {...}

```
vector<int> vi2(5);          // vektorn {0, 0, 0, 0, 0}
vector<double> vd2(2,1.2); // vektorn {1.2, 1.2}
vector<double> vd3{2,1.2}; // vektorn {2, 1.2}
```

Kopiering (konstruktor och tilldelning) och storlek

```
vector<int> vi3(vi2);         // vi3 blir kopia av vi2
vi1 = vi3;                    // vi1 blir kopia av vi3
cout << vi3.size() << endl; // skriver ut 5
vd2.resize(4);                // vd2 blir {1.2, 1.2, 0.0, 0.0}
bool tom = vd1.empty();       // ger tom = true
```

Klasserna vector och deque
Medlemsfunktioner push och pop

push lägger till ett element, storleken ökar
pop tar bort ett element, storleken minskar

`*_back` opererar på slutet, finns i båda

```
void push_back (const value_type& val);     //copy
void push_back (value_type&& val);          //move (C++11)

void pop_back();
```

bara i deque: `*_front`

```
void push_front (const value_type& val);    //copy
void push_front (value_type&& val);         //move  (C++11)

void pop_front();
```


Klasserna vector och deque
Medlemsfunktioner push och pop

Exempel

```
vector<int> v{1,1,1};
v.at(1) = 2;        // {1, 2, 1}
v.push_back(9);     // {1, 2, 1, 9}
v.push_back(4);     // {1, 2, 1, 9, 4}
v.pop_back();       // {1, 2, 1, 9}

deque<int> d;
d.push_back(3);     // d blir {3}
d.push_front(2);    // {2, 3}
d.push_front(1);    // {1, 2, 3}
d.pop_back();       // {1, 2}
d.pop_front();      // {2}
```


Klasserna vector och deque
Medlemsfunktionen assign

Tre varianter:

fill: n stycken element med samma värde

```
void assign (size_type n, const value_type& val);
```

initializer list

```
void assign (initializer_list<value_type> il);
```

range: kopierar från `first` till `last` (exkl `last`, $[first, last)$)

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last);
```


Klasserna vector och deque
Medlemsfunktionen assign, exempel

```
vector<int> v;
int a[]{0,1,2,3,4,5,6,7,8,9};

v.assign(3,1);
print_seq(v);            length = 3: [1][1][1]

v.assign({11,13,15,17});
print_seq(v);            length = 4: [11][13][15][17]

v.assign(a, a+4);
print_seq(v);            length = 4: [0][1][2][3]

std::deque<int> d;
d.assign(v.begin(), v.end());
print_seq(d);            length = 4: [0][1][2][3]
```

Exempel på iteratorer

Iteratorer

Iterator

”Pekarliknande” variabel som används för att genomlöpa en struktur (datasamling)

Exempel

```
vector<double> v(4);
vector<double>::iterator it;
for (it=v.begin(); it != v.end(); ++it)
    *it = 0;

// Ekvivalent i C++11
vector<double> v(4);
for (double &e : v)
    e = 0;
```


Iteratorer

Kategorier av iteratorer:

- Input Iterator (++ == !=) (dereferens som *rvalue*: *a, a->)
- Output Iterator (++ == !=) (dereferens som *lvalue*: *a=t)
- Forward Iterator (++)
- Bidirectional Iterator (++, --)
- Random-access Iterator (+=, -=, a[n], <, <=, >, >=)

Olika iteratorer för en containertyp (`con` symboliserar någon av containertyperna `vektor`, `deque` eller `list` med elementtypen `T`)

con<T>::iterator	löper framåt
con<T>::const_iterator	löper framåt, endast för avläsning
con<T>::reverse_iterator	löper bakåt
con<T>::const_reverse_iterator	löper bakåt, endast för avläsning

Iteratorer

Funktioner som returnerar iteratorer och som finns i alla containerklasser + klassen `string`

`begin()`	ger en iterator som pekar på det första elementet
`end()`	ger en iterator som pekar på ett tänkt element *efter* det sista elementet
`rbegin()`	ger en reverserad iterator som pekar på det sista elementet
`rend()`	ger en reverserad iterator som pekar på ett tänkt element *före* det första elementet

samt:

```
cbegin()     rcbegin()
cend()       rcend()
```


Iteratorer

Operationer med iteratorer som parametrar (iteratorintervallet `[i,j)` betecknar intervallet från och med `i` till och med positionen före `j`)

sekv<typ> s(i,j);	skapar en sekvens som initeras med `[i,j)`
s.assign(i,j);	tilldelar elem. i intervallet `[i,j)` till `s`
s.insert(p,e);	inför värdet `e` i positionen (iterator) `p`
s.insert(p,n,e);	inför `n` st `e` i positionen `p`
s.insert(p,i,j);	inför elem. i intervallet `[i,j)` i pos. `p`
s.erase(p);	tar bort elem. i pos. `p` från `s`
s.erase(p,p2);	tar bort elem. i intervallet `[p,p2)`från `s`

Iteratorer
Exempel: `vector::assign`, `vector::insert` och `vector::erase`

```
int a[] {1,2,3,4};
vector<int> v;
v.assign(a, a+4);
print_seq(v);          length = 4: [1][2][3][4]

v.insert(v.begin()+2, 3, 9);
print_seq(v);          length = 7: [1][2][9][9][9][3][4]

v.erase(v.begin()+5);
print_seq(v);          length = 6: [1][2][9][9][9][4]

v.erase(v.begin(),v.begin()+2);
print_seq(v);          length = 4: [9][9][9][4]
```


Klasserna `vector` och `deque`
Insättning med `insert` och `emplace`

insert: kopiering

```
iterator insert (const_iterator pos, const value_type& val);
iterator insert (const_iterator pos, size_type n,
                 const value_type& val);
template <class InputIterator>
iterator insert (const_iterator pos, InputIterator first,
                 InputIterator last);
iterator insert (const_iterator pos,
                 initializer_list<value_type> il);
```

emplace: konstruering *"in-place"*

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);

template <class... Args>
void emplace_back (Args&&... args);
```


Klasserna `vector` och `deque`
Exempel med `insert` och `emplace`

```
struct Foo {
  int x;
  int y;
  Foo(int a=0,int b=0) :x{a},y{b} {cout<<*this <<")\n";}
  Foo(const Foo& f) :x{f.x},y{f.y} {cout<<"**Copying Foo\n";}
};
std::ostream& operator<<(std::ostream& os, const Foo& f)
{
  return os << "Foo("<< f.x << ","<<f.y<<")";
}
vector<Foo> v;
v.reserve(4);
v.insert(v.begin(), Foo(17,42)); Foo(17,42))
                                 **Copying Foo
print_seq(v);  length = 1: [Foo(17,42)]
v.insert(v.end(), Foo(7,2));     Foo(7,2))
                                 **Copying Foo
print_seq(v);  length = 2: [Foo(17,42)][Foo(7,2)]
v.emplace_back();                Foo(0,0))
print_seq(v);  length = 3: [Foo(17,42)][Foo(7,2)][Foo(0,0)]
v.emplace_back(10);              Foo(10,0))
print_seq(v);  length = 4: [Foo(17,42)][Foo(7,2)][Foo(0,0)][Foo(10,0)]
```


Algoritmer

Tillgång till standardalgoritmer i C++ fås med direktivet

```
#include <algorithm>
```

Numeriska algoritmer:

```
#include <numeric>>
```

Ett 30-sidigt appendix (App C) i boken ger detaljer om varje algoritm

Algoritmer

Huvudkategorier av algoritmer (från App C)

1. Söka
2. Jämföra, genomlöpa, räkna
3. Kopiera och flytta element
4. Ändra och ta bort element
5. Generera nya data
6. Sortera
7. Operationer på sorterade datasamlingar
8. Operationer på mängder
9. Numeriska algoritmer
10. Heap-algoritmer

Algoritmer
Exempel: copy

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result);
```

Exempel:

```
vector<int> a(8,1);

print_seq(a);            length = 8: [1][1][1][1][1][1][1][1]

int b[]{5,4,3,2};

std::copy(std::begin(b), std::end(b), a.begin()+2);
print_seq(a);            length = 8: [1][1][5][4][3][2][1][1]
```


Algoritmer
Exempel: find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last,
                    const T& val);
```

Exempel:

```
vector<std::string> s{"Kalle", "Pelle", "Lisa", "Kim"};

auto it = std::find(s.begin(), s.end(), "Pelle");

if(it != s.end())
    cout << "Hittade " << *it << endl;
else
    cout << "Mislyckades"<< endl;
Hittade Pelle
```


Algoritmer
Exempel: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last,
                       UnaryPredicate pred);
```

Exempel:

```
bool is_odd(int i) { return i % 2 == 1; }

void test_find_if()
{
    vector<int> v{2,4,6,5,3};

    auto it = std::find_if(v.begin(), v.end(), is_odd);

    if(it != v.end())
      cout << "Hittade " << *it << endl;
    else
      cout << "Mislyckades"<< endl;
}
Hittade 5
```


Algoritmer
Insättningsiteratorer <iterator>

Exempel:

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

vector<int> e;
std::copy(v.begin(), v.end(), std::back_inserter(e));
print_seq(e);
                  length = 4: [1][2][3][4]
deque<int> e2;
std::copy(v.begin(), v.end(), std::front_inserter(e2));
print_seq(e2);
                  length = 4: [4][3][2][1]
vector<int> e3;
std::copy(v.begin(), v.end(), std::inserter(e3, e3.begin()));
print_seq(e3);
                  length = 4: [1][2][3][4]
vector<int> e4(2,11);
auto it =e4.begin()+1;
std::copy(v.crbegin(), v.crend(), std::inserter(e4, it));
print_seq(e4);
                  length = 6: [11][4][3][2][1][11]
```


Funktionsobjekt och transform

Funktionsobjekt är objekt som kan anropas som funktioner.
Algoritmen `transform` (från Kat. 5 i App C) kan hantera både funktionspekare och funktionsobjekt.

```
template < class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first, InputIt last, OutputIt d_first,
                    UnaryOperation unary_op );

template < class InputIt1, class InputIt2, class OutputIt,
           class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d_first, BinaryOperation binary_op );
```

Funktionsobjekt och `transform`

Exempel med funktionspekare

```
int kvad(int x) {
    return x*x;
}

vector<int> v{1, 2, 3, 5, 8};
vector<int> w; // w är tom!

transform(v.begin(), v.end(), inserter(w, w.begin()), kvad);

// w = {1, 4, 9, 25, 64}
```


Funktionsobjekt

Ett funktionsobjekt är ett objekt från en klass som har överlagrat funktionsanropsoperatorn `()`

Föregående exempel med funktionsobjekt

```
struct {
    int operator() (int x) const {
        return x*x;
    }
} sq;

vector<int> v{1, 2, 3, 5, 8};
vector<int> ww;  // ww är också tom!

transform(v.begin(), v.end(), inserter(ww, ww.begin()), sq);

// ww = {1, 4, 9, 25, 64}
```


Funktionsobjekt
Fördefinierade funktionsobjekt: `<functional>`

Funktioner:

```
plus, minus, multiplies, divides, modulus, negate,
equal_to, not_equal_to, greater, less, greater_equal,
less_equal, logical_and, logical_or, logical_not
```

Fördefinierat funktionsobjekt skapas med

```
operation<typ>()
```

t ex

```
    auto f = std::plus<int>();
```


Funktionsobjekt
Exempel: `std::plus` ur `<functional>`

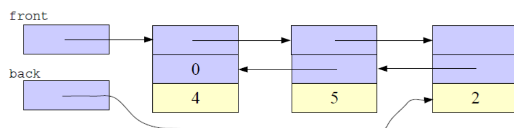
`transform` med binär funktion

```
vector<int> v1{1,2,3,4,5};
vector<int> v2(10,10);

vector<int> res2;

auto it = std::inserter(res2, res2.begin());
auto f = std::plus<int>();
std::transform(v1.begin(), v1.end(), v2.begin(), it, f);

print_seq(res2);
length = 5: [11][12][13][14][15]
```


Standardklassen `list`

- Liksom `vector` och `deque` utgör `list` en implementering av sekvenser
- Intern representation är en s.k. *länkad lista* (och inte en array som i fallen `vector` och `deque`)

```
list<int> l;
l.push_back(4); l.push_back(5); l.push_back(2);
```



Standardklassen `list`

Operationerna på en `list` är samma som på en `deque` förutom att vektorindexering (`[]` och `at()`) inte är tillåten. Utöver det finns följande operationer

l.reverse()	Vänder bak och fram på listan `l`
l.remove(e)	Tar bort alla e:n från listan `l`
l.unique()	Tar bort alla förekomster, utom den första, ur varje sammanhägande grupp av lika element i listan `l`
l.merge(l2)	Sorterar in listan `l2` i listan `l`
l.splice(p,l2)	Skjuter in elementen i listan `l2` i listan `l`, före platsen `p` (iterator). Listan `l2` blir tom.

+ varianter av vissa av dessa med fler parametrar

Avbildningar och mängder

Associativa containers

- Tabeller med söknycklar – t.ex. telefonlista med 2 kolumner (namn, telnr) där namnet utgör söknyckel
- Implementering i form av standardklasser

`map<Nyckel,Värde>`	Varje nyckel förekommer precis en gång
`multimap<Nyckel,Värde>`	En nyckel kan förekomma mer än en gång
`set<Nyckel>`	Varje nyckel förekommer precis en gång
`multiset<Nyckel>`	En nyckel kan förekomma mer än en gång

Avbildningar och mängder

<map>: `std::map`

```
map<string, int> msi;

msi.insert(make_pair("Kalle", 1));
msi.emplace("Lisa", 2);
msi["Kim"]= 5;

for(auto& a: msi) {
    cout << a.first << " : " << a.second << endl;
}

cout << msi.at("Lisa") << endl;
cout << msi.at("Kim") << endl;
```

```
Kalle : 1
Kim : 5
Lisa : 2
2
5
```


Avbildningar och mängder

<map>: `<std::multimap>`

```
multimap<string, int> mmsi;

mmsi.insert(make_pair("Kalle", 1));
mmsi.emplace("Lisa", 2);
mmsi.insert(make_pair("Kim", 5));
mmsi.emplace("Kalle", 27);
mmsi.emplace("Kim", 18);

for(auto& a: mmsi) {
    cout << a.first << " : " << a.second << endl;
}
cout << " -- find:\n";

for(auto it1 = mmsi.find("Kim");
       it1!=mmsi.end() && it1->first=="Kim"; ++it1) {
    cout << it1->first << " : " << it1->second << endl;
}
auto kalles = mmsi.equal_range("Kalle");
for(auto it = kalles.first; it != kalles.second; ++it) {
    cout << it->first << " : " << it->second << endl;
}
```

```
Kalle : 1
Kalle : 27
Kim : 5
Kim : 18
Lisa : 2
 -- find:
Kim : 5
Kim : 18
Kalle : 1
Kalle : 27
```


Avbildningar och mängder

<map>: `map` och `<multimap>`

Exempel: Nummerlogger med <telnr, antal_ggr>

```
map<string, int> numlog;
numlog.insert(make_pair(string("046-112233"), 3));
numlog.insert(make_pair(string("042-123456"), 2));
numlog.insert(make_pair(string("0413-987654"), 3));
numlog.insert(make_pair(string("042-123456"), 4));
// Sista raden ger ingen uppdatering, ty upprepning.
// Vid multimap hade däremot raden lagts till tabellen
// (troligen hade multimap varit ett bättre alternativ här)
cout << numlog.size() << endl;
```


Avbildningar och mängder

Parametrar för `std::map` och `std::multimap`

```
template < class Key,                    // map::key_type
           class T,                      // map::mapped_type
           class Compare = less<Key>,    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >
           > class map;
```

och multimap analogt

Operationer

```
insert, emplace, [], at, find, count, erase, clear,
size, empty, lower_bound, upper_bound, equal_range
```


Avbildningar och mängder

`std::set`: i princip en `std::map` med endast nycklar

Parametrar för `std::set` och `std::multiset`

```
template < class T,                   // set::key_type
           class Compare = less<T>,   // set::key_compare
           class Alloc = allocator<T> // set::allocator_type
           > class set;
```

och multiset analogt

Operationer

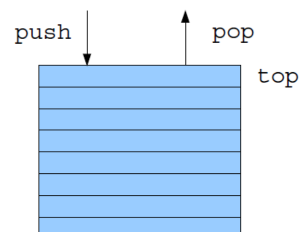
```
insert, find, count, erase, clear, size, empty,
lower_bound, upper_bound, equal_range
```

Köer och stackar

- Förenklade standardklasser, s.k. *adapterklasser*, implementerade med hjälp av någon av de andra standardklasserna:
  `stack`, `queue`
- Enklare gränssnitt med färre operationer
- Kan inte använda iteratorer

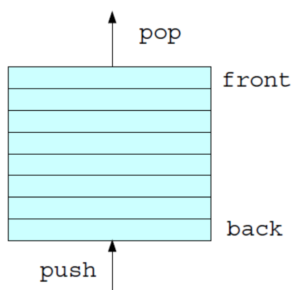
Köer och stackar

- Stack: LIFO-struktur (Last In First Out)
- Operationer: `push`, `pop`, `top`, `size` och `empty`



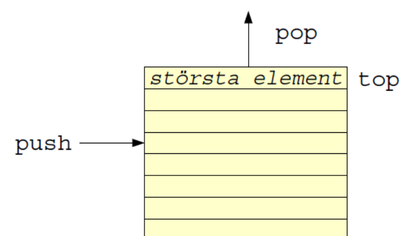
Köer och stackar

- Kö: FIFO-struktur (First In First Out)
- Operationer: `push`, `pop`, `front`, `back`, `size` och `empty`



Köer och stackar

- Prioritetskö: Som kö fast elementen har prioritet. Elementet med högst prioritet ligger först i kön.
- Operationer: `push`, `pop`, `top`, `size` och `empty`



Köer och stackar

Exempel: Använda stack för baklängesutskrift

```
#include <stack>
#include <iostream>
using namespace std;

int main () {
    stack<char> s;
    char c;
    cout << "Skriv in text och avsluta med <CR>";
    while ((c = cin.get()) != '\n')
        s.push(c);
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }
}
```


Köer och stackar

Exempel: Lägga in heltal i kö och skriva ut dem

```
#include <queue>
#include <iostream>
using namespace std;

int main () {
    queue<int> q;
    int i;
    cout << "Skriv in tal och avsluta med Ctrl-Z" << endl;
    while (cin >> i)
        q.push(i);
    while (!q.empty()) {
        cout << q.front() << ' ';
        q.pop();
    }
}
```

Köer och stackar

Exempel: Skriva ut tal i storleksordning

```
#include <queue>
#include <iostream>
#include <utility>
using namespace std;
int main () {
    priority_queue<int> p;
    int i;
    cout << "Skriv in tal och avsluta med Ctrl-Z" << endl;
    while (cin >> i)
        p.push(i);
    while (!p.empty()) {
        cout << p.top() << ' ';
        p.pop();
    }
}
```


Vi har talat om

- Datastrukturer
- Iteratorer
- Algoritmer

Nästa föreläsning:

Vi kommer att studera länkade strukturer

- Listor
- Träd