

Programmering i C++
EDA623
Containerklasser och algoritmbibliotek

Innehåll

- Containerklasser
- Klasserna `vector` och `deque`
- Iteratorer
- Algoritmer
- Funktionsobjekt
- Standardklassen `list`
- Avbildningar och mängder
- Köer och stackar

Containerklasser – Klassificering

Sekvenser

- vector
- deque
- list

S. k. adapterklasser implementerade med ovanstående klasser

- queue
- priority_queue
- stack

Associativa containers

- set
- multiset
- map
- multimap

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(n, e), v.resize(n, e)
```

Ytterligare operationer i klassen deque

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

Klasserna vector och deque

Typen av element anges innanför hakar <>

Exempel

```
vector<double> vd1;           // vektor med flyttal
vector<int> vi1;             // vektor med heltal
// Både vd1 och vi1 blir tomma (0 element)
vector<int> vi2(5);          // vektorn {0, 0, 0, 0, 0}
vector<double> vd2(2,1.2);   // vektorn {1.2, 1.2}
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

Fler exempel

```
vector<int> v;  
v.assign(3, 1);    // v blir {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}
```

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;
```


Olika typer av iteratorer *con* symboliserar någon av containertyperna vektor, deque eller list

`con::iterator<typ>`

löper framåt

`con::const_iterator<typ>`

löper framåt, endast för avläsning

`con::reverse_iterator<typ>`

löper bakåt

`con::const_reverse_iterator<typ>`

löper bakåt, endast för avläsning

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

<code>begin()</code>	ger en iterator som pekar på det första elementet
<code>end()</code>	ger en iterator som pekar på ett tänkt element <i>efter</i> det sista elementet
<code>rbegin()</code>	ger en reverserad iterator som pekar på det sista elementet
<code>rend()</code>	ger en reverserad iterator som pekar på ett tänkt element <i>före</i> det första elementet

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

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

Exempel med iteratorer som paramstrar

```
int a[] = {5, 6, 7, 8, 9};  
vector<int> v(a, a+4);    // v = {5, 6, 7, 8}  
list<int> l(v.begin(), v.end()); // l = {5, 6, 7, 8}  
l.assign(a, a+2);        // l = {5, 6}  
list<int>::iterator it = l.begin();  
l.insert(it, 2, 4);      // l = {4, 4, 5, 6}  
l.insert(l.begin(), 3); // l = {3, 4, 4, 5, 6}  
it = l.begin();  
l.erase(++it);          // l = {3, 4, 5, 6}  
l.insert(l.begin(), v.begin(), v.end());  
                        // l = {5, 6, 7, 8, 3, 4, 5, 6}
```

Tillgång till standardalgoritmer i C++ fås med direktivet

```
#include <algorithm>
```

Numeriska algoritmer ingår inte i detta bibliotek utan kräver direktivet

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

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

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

Exempel från kategorierna 3 resp. 1: copy, find

```
int a[] = {5, 6, 7, 8, 9};  
vector<int> v(a, a+4);           // v = {5, 6, 7, 8}  
deque<int> d(6, 1);             // d = {1, 1, 1, 1, 1, 1}  
copy(v.begin(), v.end(), d.begin()); // d = {5, 6, 7, 8, 1, 1}  
  
if (find(d.begin(), d.end(), 8) != d.end())  
    cout << "finns" << endl;  
// d.end() är pos. efter d's sista element
```

Insättningsiteratörer

```
#include <iterator>
back_inserter, front_inserter, inserter
```

Exempel

```
int a[] = {4, 5, 6, 7, 8, 9};
list<int> e; //Obs! e är tom
copy(a, a+4, back_inserter(e));
// e = {4, 5, 6, 7}
copy(a+2, a+6, front_inserter(e));
// e = {9, 8, 7, 6, 4, 5, 6, 7} (Obs! ordningen)
list<int>::iterator it=e.begin();
it++;
copy(a+1, a+3, inserter(e, it));
// e = {9, 5, 6, 8, 7, 6, 4, 5, 6, 7}
```


Funktionsobjekt

Funktionsobjekt är (som man kan misstänka) släkt med funktionspekare. Algoritmen `transform` (från Kat. 5 i App C) kan hantera båda varianterna.

Först visas ett exempel med funktionspekare

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

```
int a[] = {1, 2, 3, 5, 8};  
vector<int> v(a, a+5);           // 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 ()

Fortsättning på föregående exempel fast med funktionsobjekt

```
class Kvadrerare {
public:
    int operator() (int x) const {
        return x*x;
    }
};
```

```
Kvadrerare k; // k(x) returnerar x*x
int a[] = {1, 2, 3, 5, 8};
vector<int> v(a, a+5); // v = {1, 2, 3, 5, 8}
vector<int> ww; // ww är också tom!
transform(v.begin(), v.end(), inserter(ww, ww.begin()), k);
// ww = {1, 4, 9, 25, 64}
}
```

Funktionsobjekt

Fördefinierade funktionsobjekt – kräver direktivet

```
#include <functional>
```

Fördefinierat funktionsobjekt skapas med

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

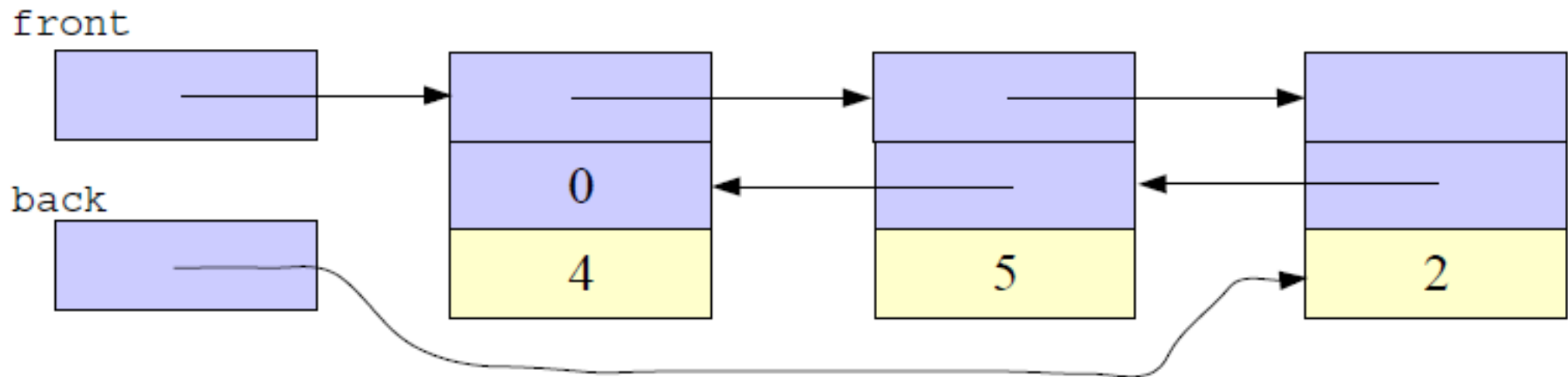
där *operation* är något av följande

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

Standardklassen list

- Liksom vector och deque utgör list en implementering av sekvenser
- Intern representation är en s.k. *länkad lista* (och inte fält 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 sammanhängande 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

Associativa containers

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

<code>map</code>	<i>nyckel, värde</i>	Varje nyckel förekommer precis en gång
<code>multimap</code>	<i>nyckel, värde</i>	Varje nyckel kan förekomma mer än en gång
<code>set</code>	<i>nyckel</i>	Varje nyckel förekommer precis en gång
<code>multiset</code>	<i>nyckel</i>	Varje nyckel kan förekomma mer än en gång

Direktiv för klasserna map och multimap

```
#include <map>
```

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

Operationer för map och multimap

<code>m<ktyp, vtyp, ctyp></code>	Anger tabelltyp (mtyp) med nyckeltyp ktyp, värdetyp vtyp jämförartyp ctyp (funktionsobjekt)
<code>m<ktyp, vtyp></code>	Som ovan men ctyp blir <code>less<ktyp></code>
<code>mtyp x;</code>	Skapar tom tabell. Söknycklar jämförs med <code>ctyp()</code>
<code>mtyp x(c);</code>	Skapar tom tabell. Söknycklar jämförs med <code>c</code>
<code>x.insert(p);</code>	Lägger in paret <code>p</code> .
<code>x[key] = value;</code>	Lägger in paret <code><key, value></code> .

Ytterligare operationer

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

Avbildningar och mängder

Operationer för set och multiset

<code>s<etyp, ctyp></code>	Anger mängdtyp (styp) med jämförartyp ctyp (funktionsobjekt)
<code>s<etyp></code>	Anger mängdtyp (styp) med jämförartyp ctyp = less<etyp>
<code>styp x;</code>	Skapar tom mängd. Elementen jämförs med ctyp()
<code>styp x(c);</code>	Skapar tom mängd. Elementen jämförs med c
<code>x.insert(p);</code>	Lägger in värdet v av typen etyp i x.

Ytterligare operationer

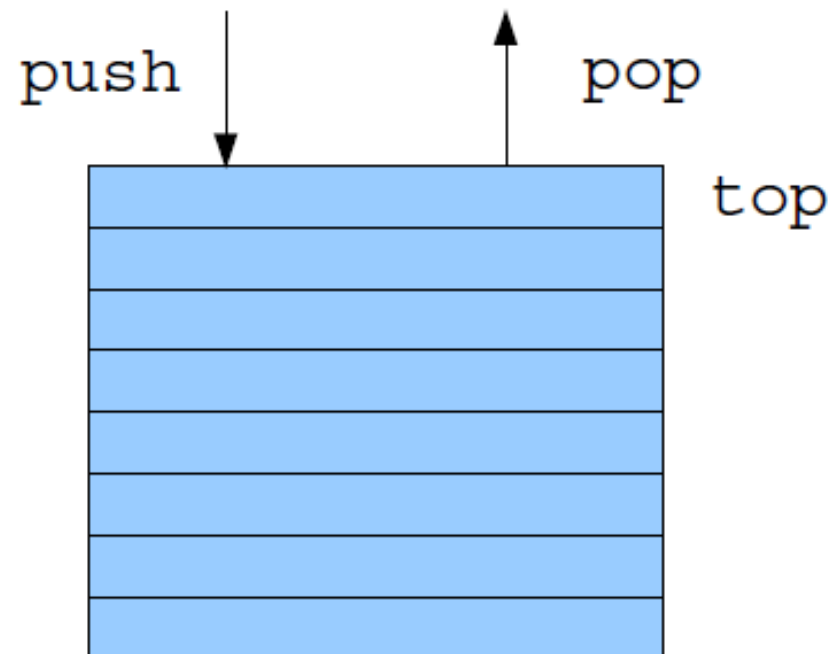
find, count, erase, clear, size, empty,
lower_bound, upper_bound, equal_range

Exempel med mängder

```
int a[] = {3, 8, 2, 0, 9, 6, 3, 4, 5};
set<int> s1(a, a+4);
// s1 innehåller {0, 2, 3, 8}
set<int> s2(a+3, a+6);
// s2 innehåller {0, 6, 9}
s2.insert(1); s2.insert(3);
// s2 innehåller {0, 1, 3, 6, 9}
copy(s1.begin(), s1.end(), inserter(s2, s2.begin()));
// s2 innehåller {0, 1, 2, 3, 6, 8, 9}
set<int> s3(a+5, a+9);
// s3 innehåller {3, 4, 5, 6}
set_intersection(s2.begin(), s2.end(), s3.begin(),
                 s3.end(), inserter(s1, s1.begin()));
// s1 innehåller {0, 2, 3, 6, 8}
```

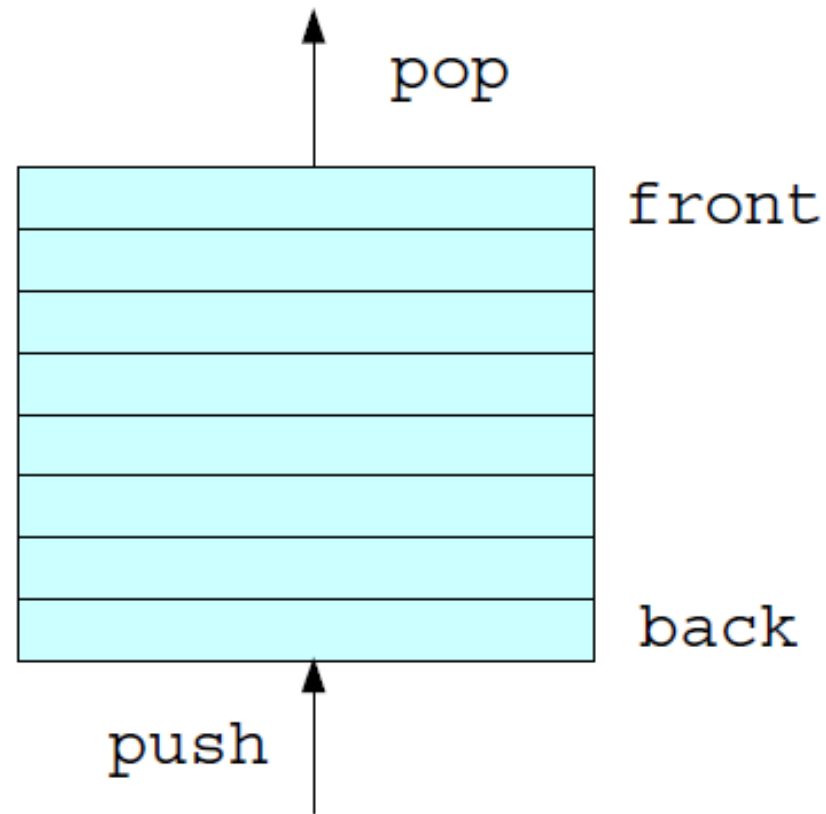
- Förenklade standardklasser, s.k. *adapterklasser*, implementerade med hjälp av någon av de andra standardklasserna:
stack, queue
- Enklare gränssnitt med färre operationer
- Kan inte använda iteratorer

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



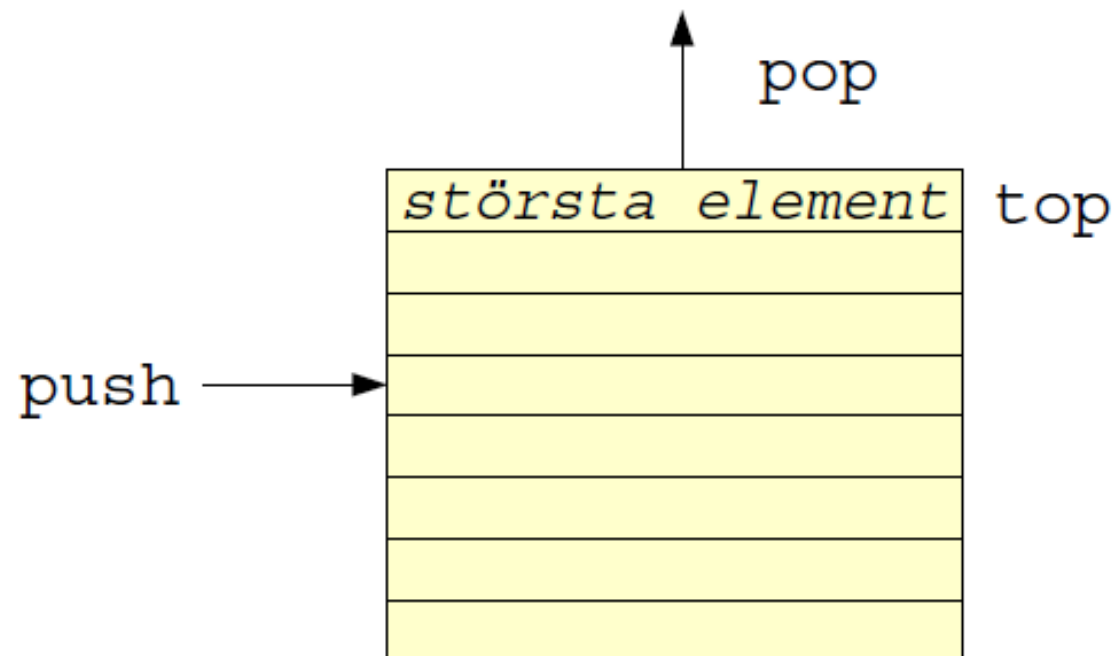
Köer och stackar

- 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



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();
    }
}
```

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();
    }
}
```


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();
    }
}
```