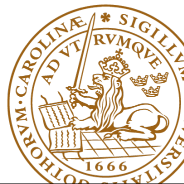
EDAF30 – Programmering i C++

5. Algoritmer. Strömmar och filer

Sven Gestegård Robertz
Datavetenskap, LTH

2016



Innehåll

1. Algoritmer
   - Standardbibliotekets algoritmer
   - Insättningsiteratorer
   - Funktionsobjekt
2. Strömmar och filer
   - Ström-klasser
   - Filer
3. Iteratorer
   - ström-iteratorer
   - Olika sorters iteratorer

Algoritmer

Tillgång till standardalgoritmer i C++ fås med direktivet

```
#include <algorithm>
```

Numeriska algoritmer:

```
#include <numeric>
```

Appendix A.2 i boken ger en översikt.

Algoritmer

Huvudkategorier av algoritmer

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

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
count och count_if

Räkna element, ur en datastruktur, som uppfyller något vilkor

- std::count(first, last, value)
  - element lika med value
- std::count_if(first, last, predicate)
  - element för vilka predicate ger true

Algoritmer
Exempel: copy och copy_if

```
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]

vector<int> b{5,4,3,2};

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

copy_if *analogt med tidigare*

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]

std::copy(v.begin(), v.end(), std::back_inserter(e2));
print_seq(e2);         length = 8: [4][3][2][1][1][2][3][4]

std::vector<int> w;
std::copy_if(v.begin(), v.end(), std::back_inserter(w), is_odd);
print_seq(w);          length = 2: [1][3]
```


Funktionsobjekt och transform

Funktionsobjekt är objekt som kan anropas som funktioner.

- funktionspekare
- funktionsobjekt (*"functor"*)

Algoritmen transform 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 square(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()), square);

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


Funktionsobjekt

Ett funktionsobjekt är ett objekt från en klass som har en **operator()**

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]
```

Exempel med accumulate <numeric>

```
auto mul = std::multiplies<int>();
int prod = std::accumulate(v1.begin(), v1.end(), 1, mul);

cout << "product(v1) = " << prod << endl;
product(v1) = 120
```


Strömmar och filer

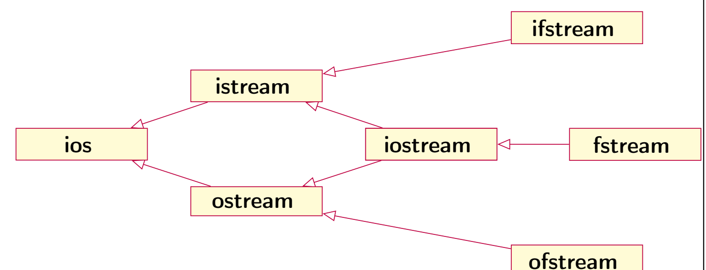
Innehåll

- Klassen ios
- Läsning av strömmar
- Utskrift av strömmar
- Koppling av filer till strömmar
- Direktaccess

Klassen ios

- Ström (stream) = Följd av tecken (bytes)
- Grundläggande klass för strömmar: ios

Klassträd för strömmar



Klassen ios

- Fördefinierade strömmar deklarerade i <iostream>: cin, cout, cerr, och clog
- Klasser deklarerade i inkluderingsfilen: <fstream>: ifstream, ofstream, fstream

Läsning av strömmar

Strömmar från klassen istream eller från subklass till denna

- Oformaterad inmatning: Läser data från strömmen utan att konvertera till annat format
  – Görs via medlemsfunktioner
- Formaterad inmatning: Data från strömmen görs om till annat format (enligt typen på variablen som ska tilldelas)
  – Görs med operatorn >>

Formaterad inmatning

Manipulatorer vid inmatning:

setw(n)	Max-antalet tecken i inläsningssträngen
ws	Hoppa fram till nästa icke-vita tecken
skipws	Hoppa över inl. vita tecken vid anv av >>
noskipws	Hoppa ej över inl. vita tecken vid anv av >>
dec	Tolka följande heltal som decimalt
oct	Tolka följande heltal som oktalt
hex	Tolka följande heltal som hexadecimalt
boolalpha	Indata för **bool** på formen **false** / **true**
noboolalpha	Indata för **bool** på formen `0` / `1`

Formaterad inmatning

Formaterad inmatning med >> och manipulatorer

```
#include <iomanip>

int i, j, k;
cout "Ange tre heltal: "
cin >> oct >> i >> hex >> j >> k;
cout << i << " " << j << " " << k << endl;

// In- och utmatning
Ange tre heltal:
12 34 56
10 52 56
```


Oformaterad inmatning

Oformaterad inmatning med medlemsfunktioner

`int get()`	Läser in och returnerar nästa tecken
`get(char& c)`	Läser in tecken till `c`
`getline(char* s, n, t)`	Läser `n` tecken till `s` med `t` som radseparator
`get(char* s, n, t)`	Som getline men sep. `t` läses ej
`read(char* s, n)`	Läser in `n` st tecken till `s`
`gcount()`	Anger antalet tecken vid senaste inläsn.
`ignore(n, t)`	Hoppar över max `n` st tecken eller till första `t`
`peek()`	Returnerar nästa tecken (som förblir oläst)
`putback(c)`	Lägger tillbaka `c` i strömmen
`unget()`	Lägger tillbaka senast lästa tecken

Klassen `ios`
Statusflaggor

Flaggor definierade i `ios`, vilka beskriver en ströms tillstånd:

`failbit`	Senaste operationen misslyckades
`eofbit`	Ett filslut påträffades vid senaste operationen
`badbit`	Ett allvarligare fel av intern art har inträffat

Klassen `ios`
Statusflaggor

Medlemsfunktioner i klassen `ios`:

`void clear();`	Slår av alla tillståndsflaggorna
`bool good();`	Ger **true** om alla flaggor **false**
`bool fail();`	Ger **true** om `failbit` el. `badbit` satt
`bool eof();`	Ger **true** om `eofbit` är satt
`bool bad();`	Ger **true** om `badbit` är satt
`bool operator!();`	Ger resultatet `fail()`

cast av en stream `s` till **bool** ger `!s.fail()`

Klassen `ios`
Inläsning tecken för tecken

OBS! `istream::eof()` sätts när man *har försökt läsa* `EOF`.

- När `eof` sätts tilldelar inte `get(char& c)` ut-parametern `c`
- men `int get()` returnerar `EOF` (`== -1`)

Utskrift till strömmar

Formaterad utmatning med <<

```
cout << uppercase << "hej svejs" << endl;
```

ger utskriften

```
hej svejs
```

```
cout << scientific << 123456789.0 << endl;
cout << uppercase << scientific << 123456789.0 << endl;
```

ger utskriften

```
1.234568e+08
1.234568E+08
```


Utskrift till strömmar

Manipulatorer vid utmatning:

setw(n)	Sätter minimalt antal positioner
setfill(c)	Anger tecken för utfyllnad (padding)
setprecision(n)	Sätter antalet decimaler (om `fixed`) eller antalet sign. siffror (om `scientific`)
boolalpha	`bool` på formen `false` / `true`
endl	Lägger in radslut i strömmen
flush	Tömmer utskriftsbufferten
setbase(n)	`setbase(16)`, `setbase(8)` osv
hex	Utskrift som hexadecimalt tal
oct	Utskrift som oktalt tal
dec	Utskrift som decimalt tal
uppercase	Ger stort `E` vid flyttalsutskrift
fixed	Utskrift med fixnotation
scientific	Utskrift med flytnotation

Utskrift till strömmar

Oformaterad utmatning med medlemsfunktioner

`put(char c)`	Skriv ut tecknet `c` till strömmen
`write(const char* s, streamsize n)`	Skriv ut `n` tecken från `s`
`flush()`	Töm utskriftsbufferten

Koppling av filer till strömmar

Öppnande av fil för läsning

```
ifstream infil("infilen.txt");

// Alt.
ifstream infil;
infil.open("infilen.txt");
```

Öppnande av fil för skrivning

```
ofstream utfil("utfilen.txt");

// Alt.
ofstream utfil;
utfil.open("utfilen.txt");
```


Koppling av filer till strömmar

Stängning av fil

```
infil.close();

utfil.close();
```

Ofta behövs inte `close` användas eftersom destruktorerna för `ifstream` och `ofstream` automatiskt anropas då den associerade strömmen destrueras

Koppling av filer till strömmar

Kopiering av fil

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

main (int argc, char* argv[]) {
        if (argc != 3) {
                cout << "Syntax: " << argv[0]
                     << " from_file to_file" << endl;
        }
        char c;
        ifstream f1(argv[1], ios::binary);//Binärfil
        ofstream f2(argv[2], ios::binary);

        while (f1.get(c)){
                f2.put(c);
        }
}
```

Koppling av filer till strömmar

Filflaggor i klassen `ios`

in	Filen skall existera och vara läsbar.
out	Om filen existerar skall den skrivas över. Om filen inte finns skall en ny skrivbar fil skapas.
app	Om filen existerar skall skrivning läggas till i slutet. Om filen inte finns skall en ny skrivbar fil skapas.
trunc	Om filen redan finns skall den skrivas över
ate	Efter öppning flyttas filpekaren till slutet av filen
binary	Filen skall hanteras som en binärfil

Kombination av flaggor med `operator|` (bitvis eller)

```
ofstream filen("fil.dat",ios::trunc|ios::binary);
```


`<stringstream>` : strängar som strömmar

```
std::stringstream ss;

ss << "Hello, string!\n";
std::cout << ss.str();

ss.str("Brave new string");

while(ss) {
    std::string s;
    ss >> s;
    std::cout << s << std::endl;
}
```

```
Hello, string!
Brave
new
string
```

Hämta/ändra innehållet i strängen med

- `string stringstream::str() const;`
- `void stringstream::str (const string& s);`

Tips: Använd `stringstream` för att enkelt experimentera med strömmar, eller skriva tester utan konsoll-I/O.

`istream_iterator<T>`

`istream_iterator<T>` : konstruktorer

```
istream_iterator(); // ger en end-of-stream istream iterator
istream_iterator (istream_type& s);
```

```
#include <iterator>

stringstream ss{"1 2 12 123 1234\n17\n\t42"};

istream_iterator<int> iit{ss};
istream_iterator<int> iit_end;

while(iit != iit_end) {
    cout << *iit++ << endl;
}
```

```
1
2
12
123
1234
17
42
```


`istream_iterator<T>`

Användning för att initiera `vector<int>`:

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<double> iit{ss};
istream_iterator<double> iit_end;

vector<int> v{iit, iit_end};

for(auto a : v) {
    cout << a << " ";
}
cout << endl;
```

```
1 2 12 123 1234 17 42
```


`istream_iterator` Felhantering

```
stringstream ss2{"1 17 kalle 2 nisse 3 pelle\n"};
istream_iterator<int> iit2{ss2};
while(!ss2.eof()) {
  while(iit2 != iit_end) { cout << *iit2++ << endl; }
  if(ss2.fail()){
    ss2.clear();
    string s;
    ss2 >> s;
    cout << "ss2: not an int: " << s << endl;
    iit2 = istream_iterator<int>(ss2); // create new iterator
  }
}
cout << boolalpha << "ss2.eof(): " << ss2.eof() << endl;
```

```
1
17
ss2: not an int: kalle
2
ss2: not an int: nisse
3
ss2: not an int: pelle
ss2.eof(): true
```

- vid fel sätts `fail`-biten för strömmen
- iteratorn sätts till `end`
- om man ändrar strömmen måste man skapa en ny iterator

Iteratorers giltighet

Generellt, om man ändrar strukturen en iterator refererar in i *blir iteratorn ogiltig*. Exempel:

- insättning
  - sekvenser
    - `vector`, `deque`* : alla iteratorer blir ogitiga
    - `list` : iteratorer påverkas inte
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- borttagning
  - sekvenser
    - `vector` : iteratorer *efter* de borttagna elementen blir ogitiga
    - `deque` : alla iteratorer blir ogitiga (i princip*)
    - `list` : iteratorer till de borttagna elementen blir ogiltiga
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- storleksförändring (`resize`): som insättning/borttagning

ostream_iterator och algoritmen copy

ostream_iterator

```
ostream_iterator (ostream_type& s);
ostream_iterator (ostream_type& s, const char_type* delimiter);
```

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<double> iit{ss};
istream_iterator<double> iit_end;

cout << fixed << setprecision(2);
ostream_iterator<double> oit{cout, " <-> "};

std::copy(iit, iit_end, oit);
```

```
1.00 <-> 2.00 <-> 12.00 <-> 123.00 <-> 1234.00 <-> 17.00 <-> 42.00 <->
```


Iteratorer

Kategorier av iteratorer:

- Input Iterator (++ == !=) (dereferens som *rvalue*: *a, a->)
- Output Iterator (++ == !=) (dereferens som *lvalue*: *a=t)
- Forward Iterator (Input- och Output Iterator, återanvändning)
- Bidirectional Iterator (som Forward Iterator, samt --)
- 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

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

- Algoritmer 10 – 10.3.1, 10.5
- Iteratorer 10.4
- Funktions-objekt 14.8
- Strömmar och filer Kapitel 8
- Ström-iteratorer 10.4.2
- Formatterad I/O 17.5.1
- Oformatterad I/O 17.5.2

Direktaccess

Direkt indexering av filposition där index är av typen streampos (heltalstyp)

is.tellg()	Ger aktuell position i inströmmen is
os.tellp()	Ger aktuell position i utströmmen os
is.seekg(pos)	Sätter aktuell position i is resp. os
os.seekp(pos)	till pos (av typen streampos)
is.seekg(off, dir)	Sätter positionen till off+dir där
os.seekp(off, dir)	off är en (ev neg.) offset och dir är ios::beg (start), ios::end (slut) eller ios::cur (akt. pos.)

<stringstream> : strängar som strömmar
Exempel: seek

```
void print_pos(stringstream& ss) {
    cout<<"tellg: "<<ss.tellg()<<", tellp: "<<ss.tellp()<< endl;
}

stringstream ss{"1234567890abcdef"};
ss.seekg(5);
cout<<"peek: "<<char(ss.peek())<<endl;    peek: 6
ss.seekg(-1,ios::end);
cout<<"peek: "<<char(ss.peek())<< endl;   peek: f
ss.seekg(ios::beg);
cout<<"peek: "<<char(ss.peek())<< endl;   peek: 1
ss.seekg(2, ios::beg);
cout<<"peek: "<<char(ss.peek())<< endl;   peek: 3
ss.seekg(2, ios::cur);
cout<<"peek: "<<char(ss.peek())<< endl;   peek: 5
print_pos(ss);                            tellg: 4, tellp: 0
ss << "XYZ";    // Skriver över
print_pos(ss);                            tellg: 4, tellp: 3
cout << "str(): " << ss.str() << endl;    str(): XYZ4567890abcdef
ss.seekp(0, ios::end);
ss << "ABC";    // Lägger till
print_pos(ss);                            tellg: 4, tellp: 19
cout << "str(): " << ss.str() << endl;    str(): XYZ4567890abcdefABC
```