

# Kapitel 4

Introduktion till mallar  
STL

# Kapitel 4 - Klassmallar, funktionsmallar och STL

- Funktionsmallar
- Klassmallar
- Mallar för medlemmar
- Specialisering
- Standardbiblioteket för mallar (STL):
  - Namnrymden `std`
  - `std::string`
  - STL-behållare (`vector`, `list`, `map`, `deque`, `set`, `multimap`, `multiset`)
  - Iteratorer, algoritmer, funktionsobjekt

# Mallar

Funktionsmallar

Klassmallar

Medlemsfunktioner  
som mallfunktioner

# Funktionsmallar

- Antag att du har två likadana funktioner med den enda skillnaden att de arbetar på olika typer (t.ex. double och int)
- Ett bättre alternativ är då **funktionsmallar** (*function templates*)
- Typen blir en parameter till funktionen och bestäms vid kompileringstillfället

# Funktionsmallar

- Typen är en parameter  $T$
- Gränssnittet för alla typer  $T$  måste klara allt som funktionen begär, t.ex. `+`, `=` och anrop till en funktion `print()`
- Detta medför att man kan skriva en funktion ovetandes om vilken typ den ska arbeta på, så länge typen uppfyller kraven på gränssnittet
- Man talar om polymorfi vid kompileringstillfället

# Funktionsmallar

## Ett första exempel på funktionsmall

```
class A {      // A är en klass som klarar av att jämföras
public:
    bool operator<(const A &a) const { return i < a.i; }
    int i;
};

template<class T> // funktionen max använder operator<
const T &max(const T &t1, const T &t2) {
    return t1 < t2 ? t2 : t1;
}

A a, b;
max(a, b);
max(2.14, 3.14);
```

# Funktionsmallar

## Ett mer avancerat exempel på funktionsmall

```
struct A      { virtual ~A() {} };
struct B : A  {};
struct C : A  {};

template<class T> // generisk factoryfunktion
A *create()      // returnerar pekare till basklass
{
    return new T; // skapa instans av typ T
}

A* (*fp)();      // funktionspek, inga arg, returnera A*

fp = create<B>;  // peka på B-factory
A *b = fp();    // skapa ett B-objekt

fp = create<C>;  // peka på C-factory
A *c = fp();    // skapa ett C-objekt
```

# Klassmallar

För att skapa en vektorklass som innehåller godtycklig typ använder man en klassmall (*class template*):

```
// i fil array.h
template<class T>
class Array
{
public:
    Array() {}
    const T &operator[](int i) const; // this är konstant
    T & operator[](int i);         // this är ej konst
protected:
    T array[100];
};
...
```

# Klassmallar

```
// fortsättning på fil array.h
template<class T>           // obs hur klassen anges
const T& Array<T>::operator[](int i) const
{
    return array[i];
}

template<class T>           // obs hur klassen anges
T& Array<T>::operator[](int i)
{
    return array[i];
}

Array<int> a;               // instansiera
a[2] = 5;                  // ok: operator[] returnerar T&
```

# Mallar för medlemsfunktioner

Funktionsmallar fungerar lika bra för klassmedlemmar

```
class A {
public:
    template<class T>
    void foo(T t) const {
        std::cout << "type: " << typeid(T).name()
                  << std::endl;
    }
};

A a;
void *p = 0;
a.foo(7);           // type: int
a.foo(3.14);       // type: double
a.foo(p);          // type: void *
a.foo(a);          // type: class A
```

# Specialisering

Med en **specialisering** (*specialization*)  
särbehandlar man typer

```
template<class T>                // klarar alla typer
void swap(T &t1, T &t2)
{
    T tmp = t1;                  // byt plats
    t1 = t2;
    t2 = tmp;
}

template<>                       // specialisering för Vector
void swap(Vector &v1, Vector &v2)
{
    swap(v1.size, v2.size);     // byt storlek
    swap(v1.p, v2.p);           // byt pekare!
}
```

# Specialisering

## Exempel på spesialisering av medlem

```
class A
{
public:
    template<class T> void foo(T) const {
        std::cout << "Unknown type" << std::endl;
    }
    template<> void foo(int) const {
        std::cout << "type was int" << std::endl;
    }
    template<> void foo(double) const {
        std::cout << "type was double" << std::endl;
    }
};
```

# Specialisering

## Exempel på specialisering av en klass

```
// definition av vanlig mallklass  
template<class T> class A { T data; };  
  
// specialisering, denna definition skapas för pekare  
// av alla typer, T blir typen som pekas på  
template<class T> class A<T *> { T data; T *ptr; };  
  
// specialisering, denna definition skapas för void*  
template<> class A<void *> {}; // ingen typ T finns
```

# STL – Standard Template Library

Strängar  
Behållare  
Iteratorer  
Algoritmer  
Hjälpstrukturer

# Standardbiblioteket för mallar

- C++-standarden kräver att varje implementation av C++ kommer med ett stort antal mallar för klasser och funktioner.
- Biblioteket kallas *Standard template library* (STL).
- Allt innehåll i standardbiblioteket ligger i namnrymden `std`, t.ex. `std::string`

# Standardbiblioteket för mallar

Exempel på funktionalitet i STL:

- strängar (`#include <string>`) genom `std::string`
- behållarklasser (*containers*) (t.ex. `#include <vector>` för att använda `std::vector`)
- algoritmer (`#include <algorithm>`) genom t.ex. `std::sort` eller `std::foreach`
- Funktorer och hjälpklasser

# std::pair

Först, ett litet exempel på en hjälpklass

```
#include <utility>           // för pair och make_pair

int main()
{
    std::string s = "hej";
    std::pair<int, std::string> pr; // skapa objekt
    pr = std::make_pair(7, s);     // skapa nytt par

    std::cout << pr.first << ", " // åtkomst
               << pr.second << std::endl;

    return 0;
}
```

# Strängar i STL

# Strängar i STL

- Standardbiblioteket bidrar med en kraftfull strängklass `std::string`.
- `std::basic_string` är en behållare som håller bokstäver såsom `char` eller `wchar_t`
- `std::string` är en typdefinition av `std::basic_string<char>`
- För minneseffektivitetens skull har de flesta implementationer infört **referensräkning**.

# Strängar i STL

Exempel på stränganvändning:

```
#include <string>

std::string s("C++");           // skapa sträng
s += " is great!";             // lägg till sträng

std::cout << "True: "          // använd c_str() för
      << s.c_str() << std::endl; // tillgång till data
std::cout << "Predecessor: "    // använd oper[] för
      << s[0] << std::endl;     // tillgång till bokstav
std::reverse(s.begin(), s.end()); // vänd på strängen
std::cout << s << std::endl;
```

# Strängar i STL

Exempel på medlemsfunktioner:

```
basic_string& operator+=(const basic_string& s)
basic_string& insert(size_type pos,
                    const basic_string& s)
iterator erase(iterator first, iterator last)
size_type find_first_of(const basic_string& s)
bool empty() const
size_type size() const
```

# Vektorer i STL

# Vektorer i STL

- STL innehåller en mycket användbar behållare `std::vector`
- Vektorn i STL har **dynamisk allokering**, till skillnad från C-vektorer
- C++-standarden anger att vektorer skall garantera uppslagning och insättning / borttagning sist i konstant tid
- Långsamma operationer är insättning / borttagning på godtycklig plats

# Vektorer i STL

Exempel på användning av `std::vector`:

```
std::vector<int> v;           // vektorn innehåller heltal
v.push_back(3);             // sätt in sist
assert(v.size() == 1 &&    // size() ger antal element
       v.capacity() >= 1 && // capacity() ger max antal
       v.back() == 3);     // element utan omallokering
                             // hämta sista elementet

v.push_back(7);             // sätt in sist
v.pop_back();               // ta bort sista elementet

for(int i = 0; i < v.size(); i++) // skriv ut ett
    std::cout << v[i] << std::endl; // element i taget
```

# Vektorer i STL

Exempel på medlemsfunktioner:

```
void push_back(const T &)  
iterator erase(iterator pos)  
iterator insert(iterator pos, const T &)  
iterator begin()  
iterator end()  
void clear()  
void reserve(size_t)  
size_type size() const  
size_type length() const
```

# Listor i STL

# Listor i STL

- STLs lista är en implementation som har samma egenskaper som en dubbellänkad lista
- C++-standarden anger att insättning och borttagning skall ske på konstant tid
- Uppslagning av godtyckligt element är en långsam funktion eftersom man då måste iterera över elementen

# Listor i STL

Exempel på användning av `std::list`:

```
std::list<int> l;           // listan innehåller heltal
l.push_back(0);           // lägg 0 sist
l.push_front(1);          // lägg 1 först
l.insert(++l.begin(), 2); // lägg 2 efter första elem.
                          // listan innehåller nu 1 2 0
```

# Listor i STL

Exempel på medlemsfunktioner:

```
void push_front(const T &)  
void push_back(const T &)  
void pop_front()  
void pop_back()  
iterator insert(iterator pos, const T &)  
iterator erase(iterator pos)  
bool empty() const  
void clear()  
size_type size() const
```

# Avbildningar i STL

# Avbildningar i STL

- För snabb uppslagning finns en behållarklass **`std::map`**
- C++-standarden anger att uppslagning och insättning skall ske på logaritmisk tid
- **`std::map`** implementeras oftast som ett röd-svart träd
- Iteration över elementen sker i stigande nyckelordning, t.ex. i bokstavsordning för **`std::string`**

# Avbildningar i STL

## Exempel på användning av `std::map`:

```
struct comp_string {
    bool operator()(const char* s1, const char* s2) const
    { return strcmp(s1, s2) < 0; }
};

// avbildar char* till int, och ordnar
// nycklarna enligt comp_string
std::map<const char*, int, comp_string> months;

months["januari"] = 31;
months["februari"] = 28;
...
months["december"] = 31;
std::cout << "juni har " << months["june"]
           << " dagar" << std::endl;
```

# Avbildningar i STL

Exempel på medlemsfunktioner:

```
iterator find(const key_type& k)  
pair<iterator, bool> insert(const value_type& x)  
size_type erase(const key_type& k)  
void clear()  
size_type size() const
```

# Avbildningar i STL

## Varning och tips på användning

```
typedef std::map<std::string, std::string> map;
map m;

// operator[] lägger in nytt element
m["Mr. Brown"] = "Mrs. Brown";
m["Mr. Green"] = "Mrs. Green";

// operator[] lägger in nytt element även här
// och returnerar tomma strängen
if(m["Mr. Black"] == "")
    std::cout << "hoppsan" << std::endl;

// bättre: använd iteratorer
map::const_iterator it = m.find("Mr. Red");
if(it != m.end())
    std::cout << it->second << std::endl;
```

# Övriga STL-behållare

- Vi har talat om dessa:
  - **string**, **vector**, **list**, **map**
- Man bör också känna till:
  - **deque** (*double ended queue*)
  - **set** (ordnad mängd utan dubletter)
  - **multimap** och **multiset** (**map** och **set**, fast tillåter dubletter)

# Övriga STL-behållare

- Derivat som använder sig av de tidigare nämnda behållarna:
  - **stack** (först in sist ut)
  - **priority\_queue** (högst värde ut först)
- Numeriska behållare:
  - **bitset** (sträng av ettor och nollor)
  - **valarray** (matematiska vektorer)

# Hashtabeller

- En vanlig fråga är var man kan hitta hashtabeller i standardbiblioteket
- Svaret är att det inte finns några
- Dock kommer dessa förmodligen i nästa standard
- Se SGIs hemsida ([www.sgi.com/tech/stl](http://www.sgi.com/tech/stl)) för dokumentation av

`hash_map, hash_set,  
hash_multimap, hash_multiset`

# Iteratorer i STL

# Iteratorer

- För att iterera över behållare i STL finns **iteratorer**
- Syntaxen hos de överlagrade operatorerna i iteratorklassen imiterar pekarens syntax
- Iteratorer finns i varje behållarklass och heter **T::iterator** och **T::const\_iterator**

# Iteratorer

```
std::vector<int> v;  
std::vector<int>::iterator i;           // skapa iterator  
...  
  
// iterera: jämför med v.end() som är utanför vektorn  
for(i = v.begin(); i != v.end(); i++) // ++ ger nästa  
    if(*i == 7)                       // * avrefererar  
    {  
        std::cout << "Found '7' in element number "  
                  << std::distance(v.begin(), i)  
                  << std::endl;  
    }
```

# Iteratorer

Olika typer av iteratorer (iteratorhierarkin):

- `std::random_access_iterator`  
`++ -- += -= + - * < <= == > != =`
- `std::bidirectional_iterator`  
`++ -- * != =`
- `std::forward_iterator`  
`++ * != =`
- `std::output_iterator` och  
`std::input_iterator`  
`++ * !=`

Dessa är implementerade som tomma basklasser.

De används för att bestämma typ (genom t.ex. referens)

# Algoritmer och funktionsobjekt

# Algoritmer

- I STL finns många användbara algoritmer
- Samtliga algoritmer opererar på iteratorer
- Iteratorer finns för alla behållare och algoritmerna fungerar därför på alla behållare
- Detta gäller även dem man skriver själv, förutsatt att iteratorn uppfyller villkoren

# Algoritmer

```
#include <algorithm>                                // algoritmer

int a[] = {1, 4, 2, 8, 5, 7};
const int n = sizeof(a) / sizeof(int); // antal element
std::sort(a, a + n);                             // sortera
std::copy(a, a + n,                               // mata ut
          std::ostream_iterator<int>(std::cout, " "));

std::vector<int> b;
b.push_back(7);
...
std::sort(b.begin(), b.end()); // sortera
std::unique(b.begin(), b.end()); // ta bort dubletter
```

# Funktionsobjekt

- **Funktionsobjekt** (*function objekt*) är objekt som imiterar syntaxen hos en funktion
- En fördel i jämförelsen med funktionspekare är att funktionsobjekt kan innehålla data
- När man använder funktionspekare sker alla anrop genom att avreferera pekaren. Samtidigt är det svårt att göra funktionen **inline**

# Funktionsobjekt

```
struct less_abs
{
    bool operator()(double x, double y)
    { return fabs(x) < fabs(y); }
};
std::vector<double> v;
...

// konstruera less_abs-objekt
std::sort(v.begin(), v.end(), less_abs());

// rand är fkt.pekare
std::vector<int> u(100);
std::generate(u.begin(), u.end(), rand);
```

# Funktionsobjekt

Skilj på typer och objekt (instanser av typer)

```
struct A    { int i; };
struct Comp {
    bool operator()(const A &a, const A &b) const
    {
        return a.i < b.i;
    }
};

// Comp är en typ och används inuti std::map
// vid kompileringstillfället
std::map<A, A, Comp> m;

// Comp() är en instans och används i funktionen
// std::sort vid körningstillfället
std::vector<A> v;
std::sort(v.begin(), v.end(), Comp());
```