

Del 4 – Klassmallar, funktionsmallar och STL

Ämnesområden denna föreläsning:

- Funktionsmallar (*function templates*)
- Klassmallar
- Mallar för medlemmar
- Specialisering

Slide 1

- Standardbiblioteket för mallar (STL):
 - Namnrymden `std`
 - Iteratorer
 - `std::string`, `std::vector`, `std::list`, `std::map`,
`std::set`
 - Algoritmer, funktionsobjekt
- Typinformation under körning

Funktionsmallar

- Du vill skapa en funktion som kan användas med olika typer av objekt
- ### Slide 2
- Man kan använda generiska datatyper, funktionspekare och typkonvertering, problemet är att programmet inte blir typsäkert
 - Ett bättre alternativ är **funktionsmallar** (*templates*)

Slide 3

```

class A {
public:
    bool operator<(const A &a) const { return a.i < i; }
    int i;
};

template<class T>
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);

```

Klassmallar

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

Slide 4

```

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

// i fil myarray.cpp

```

Slide 5

```

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

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

...
MyArray<int, 17> a;                // instansiera
a[2] = 5;                          // ok: kallar operator[], returnerar T&

```

Mallar för medlemmar

Funktionsmallar fungerar lika bra för klassmedlemmar

Slide 6

```

class A {
public:
    template<class T> operator T() const;
    int data;
};

template<class T> A::operator T() const {
    cout << "Ny typ: " << typeid(T).name() << endl;
    return T(data);          // ok, men farligt
}

A a;
int i = a;                  // Ny typ: int
double d = a;              // Ny typ: double
void *p = a;               // Ny typ: pointer to void

```

Specialisering

Ibland vet man hur man ska hantera vissa typer speciellt. Man kan då ange en **specialisering** (*specialization*).

Slide 7

```
class A
{
public:
    template<class T> operator T() const;
    int data;
};
```

```
template<class T>
A::operator T() const {
    cout << "Unknown type" << endl;
    return 0;
}
```

Slide 8

```
template<>
A::operator int() const {
    cout << "type was int" << endl;
    return static_cast<int>(data);
}
```

```
template<>
A::operator double() const {
    cout << "type was double" << endl;
    return static_cast<double>(data);
}
```

Exempel på specialisering av en klass

```
// definition av ursprungsklassen
template<class T> class A {};

// specialisering, denna definition skapas för pekare
// av alla typer, T blir typen som pekars på
template<class T> class A<T *> {};

// specialisering, denna definition skapas för void*
template<> class A<void *> {};
```

Slide 9

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 STL ligger i namnrymden `std`. Exempel på funktionalitet:

Slide 10

- strängar (`#include <string>`), används med `std::string`
- behållarklasser (*containers*) (t.ex. `#include <vector>` som används med `std::vector`) med tilldelning och kopiering
- algoritmer (`#include <algorithm>`), används t.ex. med `std::sort` eller `std::foreach`

Strängar i STL

Standardbiblioteket bidrar med en kraftfull strängklass `std::string`. För minneseffektivitetens skull har de flesta implementationer infört **referensräkning**.

- `std::basic_string` är en behållare som håller bokstäver såsom `char` eller `wchar_t`

Slide 11 • `std::string` är en typdefinition av `std::basic_string<char>`

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) const
bool empty() const, size_type size() const
```

Exempel på stränganvändning:

```
const char *c = "C++";
string s(c);           // skapa sträng
s += " is fun";       // lägg till sträng
```

Slide 12

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

Vektorer i STL

STL innehåller en mycket användbar behållare `std::vector`.

Denna vektor, till skillnad från vektorer på stacken, har **dynamisk allokering**.

- C++-standarden anger att vektorer skall garantera uppslagning och insättning/borttagning sist i konstant tid

Slide 13

- Långsamma operationer är insättning efter/borttagning av godtyckligt element

Exempel på medlemsfunktioner:

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

Exempel på användning av `std::vector`:

```
vector<int> v; // vektorn innehåller heltal
v.insert(v.begin(), 3); // sätt in först
assert(v.size() == 1 && // size() ger antal element
       v.capacity() >= 1 && // capacity() ger max antal
       // element innan omallokering
       v[0] == 3); // hämta element 0

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

Slide 14

Iteratorer

För att iterera över behållare i STL finns **iteratorer**. Då man använder **vector** skulle man i många fall kunna använda indexoperatören [], men för effektivitetens och enhetlighetens skull har alla behållare en iteratorclass.

Slide 15

```
vector<int> v;
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 elem
    if(*i == 7)                       // * ger utpekad elem
        cout << "Found '7' in element number "
              << (i - v.begin()) << endl; // ok: differens
```

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

Slide 16

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& x)
iterator erase(iterator pos)
bool empty() const, void clear(), size_type size() const
```

Exempel på användning av `std::list`:

Slide 17

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

Avbildningar i STL

För snabb uppslagning finns en behållarklass `std::map`

- C++-standarden anger att uppslagning och insättning med nyckel skall ske på sämst logaritmisk tid.
- `std::map` är implementerad som ett träd och elementen är sorterade

Slide 18

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
```

Exempel på användning av `std::map`:

Slide 19

```

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
map<const char*, int, comp_string> months;

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

```

Algoritmer

I STL finns många användbara algoritmer. De opererar på iteratorer och eftersom iteratorer finns för alla behållare fungerar algoritmerna på alla behållare.

Slide 20

```

#include <algorithm> // algoritmer
int a[] = {1, 4, 2, 8, 5, 7};
const int n = sizeof(a) / sizeof(int); // antal element
sort(a, a + n); // sortering
copy(a, a + n, // utmatning
      ostream_iterator<int>(cout, " ")); // 1 2 4 5 7 8

std::vector<int> b;
b.push_back(7);
...
sort(b.begin(), b.end()); // sortering

```

Funktionsobjekt

När man använder funktionspekare sker alla anrop genom att avreferera pekaren. Samtidigt är det inte möjligt att göra funktionen `inline`. En bättre lösning är **funktionsobjekt**:

Slide 21

```
struct less_abs
    : public binary_function<double, double, bool> {
    bool operator()(double x, double y)
    { return fabs(x) < fabs(y); }
};
vector<double> v;
...
sort(v.begin(), v.end(), less_abs()); // konstruera
                                     // less_abs-objekt
vector<int> u(100);
generate(u.begin(), u.end(), rand); // rand är fkt.pekare
```

Typinformation under körning

- Genom operatorn `typeid()` får man en instans av klassen `type_info` som innehåller typinformation
- `type_info` har en metod `name()` som ger klassens namn

Exempel på användning:

Slide 22

```
class A {};
A *a = new A;
const type_info &tA = typeid(A); // typinfo på klass
const type_info &ta = typeid(*a); // typinfo på objekt
cout << tA.name() << endl;       // namn på typ
if(tA == ta)                    // typer kan jämföras
    cout << "They are the same" << endl;
if(ta.before(tA))               // typer har en ordning
    cout << "ta is before tA" << endl;
```