EDAF30 – Programmering i C++

6. Typer, pekare och konstanter. Funktionsmallar

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Generisk programmering
Templates (mallar)

- Använder *typparametrar* för att skriva mer generella klasser och funktioner
- Slipper manuellt skriva en ny klass/funtion för varje datatyp som ska hanteras
- statisk polymorfism
- En mall *instansieras* av kompilatorn för typer den används för
  - varje instans är en separat klass/funktion
  - vid kompilering: ingen kostnad vid exekvering

Funktionsmallar

Exempel: Vanliga minimifunktionen

```
template<class T>
const T& minimum(const T& a, const T& b) {
    if (a < b)
        return a;
    else
        return b;
}
```

Kan instansieras för alla typer som har operatorn <

Funktionsmallar

Instansiering av funktionsmallar sker i vissa fall automatiskt t.ex. vid utskrift:

```
double x = 7.0, y = 5.0;
long m = 5, n = 7;
//...
cout << minimum(x, y) << endl;
cout << minimum(m, n) << endl;

// Blandat funkar inte (ingen casting görs!)
cout << minimum(m, y) << endl;

// För att forcera casting ger man typ-parametern explicit
cout << minimum<double>(m, y) << endl;
```

En explicit instans av en funktionsmall är en vanlig funktion
⇒ implicit typkonvertering av argument

Funktionsmallar
Överlagring med vanlig funktion

```
struct Name{
    string s;
    //...
};
```

Överlagring för `Name&`

```
const Name& minimum(const Name& a, const Name& b)
{
    if(a.s < b.s)
        return a;
    else
        return b;
}
```

Funktionsmallar
Funktionsmall för minsta elementet i en array

```
template<typename T>
T& min_element(T a[], size_t n)
{
    cout << "[min_element<"<<typeid(T).name()<<"[]>]\n";
    size_t idx = 0;

    for(size_t i = 1; i < n; ++i) {
        if(a[i] < a[idx])
            idx = i;
    }
    return a[idx];
}
```

Användning (*kompilatorn härleder typparametern T*)

```
int a[] {3,5,7,6,8,5,2,4};

int ma = min_element(a, sizeof(a)/sizeof(int));
```


Funktionsmallar kan överlagras

Generalisering: överlagra `min_element` för fler datastrukturer

Funktionsmallar
Funktionsmall för minsta elementet i iterator-par

```
template<typename FwdIterator>
ForwardIterator min_element(FwdIterator start, FwdIterator end)
{
    if(start==end)
        return end;
    ForwardIterator res=start;

    auto it = start;
    while(++it != end){
        if(*it < *res)
            res = it;
    }
    return res;
}
```

- Standardalgoritmerna är templates
- Detta är en version av `std::min_element`

Användning

```
int a[] {3,5,7,6,8,5,2,4};
int ma2 = *min_element(begin(a), end(a));
int ma3 = *min_element(a+2,a+4);

vector<int> v{3,5,7,6,8,5,2,4};
int mv = *min_element(v.begin(), v.end());
```


Funktionsmallar
Funktionsmall för minsta elementet i en `vector<T>`

Parametriserad på *element-typen* för `vector`

```
template<typename T>
T& min_element(vector<T>& c)
{
    if(v.begin()==v.end())
        throw std::range_error("empty vector");
    return *min_element(v.begin(), v.end());
}
```

Användning

```
vector<int> v{3,5,7,6,8,5,2,4};
int mv2 = min_element(v);
```


Funktionsmallar
Funktionsmall för minsta elementet i en container

Variant som kan användas för en godtycklig klass som

- har `begin()` och `end()`
- har en **typedef** som definierar namnet `value_type`

(detta uppfylls av alla standard-containers)

```
template<class Container>
typename Container::value_type& min_element(Container& c)
{
    if(c.begin()==c.end())
        throw std::range_error("empty container");
    return *min_element(c.begin(), c.end());
}
```

Kan även deklareras med två typ-parametrar:

```
template<typename Container,
         typename T = typename Container::value_type>
T& min_element(Container& c)
```


Funktionsmallar
Funktionsmall för minsta elementet i en array

Trick: värdeparameter för arrayens storlek, och inferens

```
template<typename T, size_t N>
T& min_element(T (&a)[N])
{
    size_t idx = 0;

    for(size_t i = 1; i < N; ++i) {
        if(a[i] < a[idx])
            idx = i;
    }
    return a[idx];
}
```

Användning

```
int a[] {3,5,7,6,8,5,2,4};

int ma4 = min_element(a);
```

Här vet kompilatorn storleken på `a[]` och fyller i mall-parametern `N`

Funktionsmallar
minsta elementet i *nånting man kan iterera över*

Användning

```
int a[] {3,5,7,6,8,5,2,4};
int& ma = min_element(a);

vector<int> v{3,5,7,6,8,5,2,4};
int& mv = min_element(v);

deque<int> d{v.begin(), v.end()};
int& md = min_element(d);

string s{"kontrabasfiol"};
char& ms = min_element(s);
```


Funktionsmallar
`std::min_element` för typer som inte har <

Överlagring med en template-parameter: LESS (en egenskapsklass)

```
template<class IT, class LESS>
IT min_element(IT first, IT last, LESS cmp)
{
    IT m = first;
    for (IT i = ++first; i != last; ++i)
    if (cmp(*i, *m))
        m = i;
    return m;
}
```

LESS måste ha **operator**() och typerna måste stämma, t ex:

```
class Str_Less_Than {
public:
    bool operator () (const char *s1, const char *s2)
    {
        return strcmp(s1, s2) < 0;
    }
};
```


Funktionsmallar
`std::min_element` för typer som inte har <

Exempel på användning med stränglista:

```
list<const char *> tl = { "hej", "du", "glade" };
Str_Less_Than lt; // funktionsobjekt

cout << *min_element(tl.begin(), tl.end(), lt);
```

Str_Less_Than-objektet kan skapas direkt i parameterlistan:

```
cout << *min_element(tl.begin(), tl.end(), Str_Less_Than());
```

(C++11) lambda: anonymt funktions-objekt

```
auto cf= [](const char* s, const char* t){return strcmp(s,t)<0;};

cout << *min_element(tl.begin(), tl.end(), cf);
```


Strukturer
Åtkomst av **struct**-medlemmar

```
Struct Point{
    int x;
    int y;
}

Point p;

Point& rp;

Point* pp;

...

int i = p.x;     // åtkomst via namn (på variabel)

int j = rp.x;    // åtkomst via referens (alias för namn)

int k = pp->x;   // åtkomst via pekare
```


Access av medlemmar via pekare
Operatorn ->

Givet en pekare p, så kan vi uttrycka
"Medlemmen x i objektet som p pekar på" på två sätt:

- p->x
- (*p).x

Typer
Två sorters konstanter

- En variabel deklarerad **const** får inte ändras (`final` i Java)
  - Betyder ungefär "Jag lovar att inte ändra denna variabel."
  - Kontrolleras av kompilatorn
  - Används ofta för att specificera gränssnitt
    - En funktion som inte ändrar värdet på argument
    - En medlemsfunktion ("metod") som inte ändrar objektets tillstånd.
  - Viktigt vid (medlems-)funktionsöverlagring
- En variabel deklarerad **constexpr** måste ha ett värde som kan beräknas vid kompilering.
  - Används för att specificera konstanter
  - Infördes i C++-11

Medlemsfunktioner kan vara **const**

- Betyder att de inte ändrar objektets tillstånd
- Viktigt vid överlagring
  - `T func(int)` och `T func(int) const` är olika funktioner

exempel:

```
class Container{
public:
  double& operator[]();
  double  operator[]() const;
};
```


Funktioner kan vara **constexpr**

- Betyder att de kan beräknas vid kompilering om argumenten är **constexpr**

exempel:

```
constexpr int square(int x)
{
    return x*x;
}

void test_constexpr_fn()
{
    char matrix[square(4)];

    cout << "sizeof(matrix) = " << sizeof(matrix) << endl;
}
```

Utan **constexpr** får man felet

```
error: variable length arrays are a C99 feature
```


const och pekare

const modifierar allt som står till vänster om (undantag: om **const** står först modifieras det som kommer omedelbart efter)

Exempel

```
     int* ptr;
const int* ptrToConst;  //NB! (const int) *
int const* ptrToConst,  // ekvivalent, tydligare?

int* const  constPtr;   // pekaren är konstant

const int* const constPtrToConst; // Både pekare och värde
int const* const constPtrToConst; // ekvivalent, tydligare?
```

Var noggrann när du läser:

```
char *strcpy(char *dest, const char *src);
```

`(const char)*, inte const (char*)`

const och pekare
Exempel

```
void Exempel( int* ptr,
              int const * ptrToConst,
              int* const constPtr,
              int const * const constPtrToConst )
{
    *ptr = 0;                   // OK: ändrar innehållet
    ptr  = nullptr;             // OK: ändrar pekaren

    *ptrToConst = 0;            // Fel! kan inte ändra innehållet
    ptrToConst  = nullptr;      // OK: ändrar pekaren

    *constPtr = 0;              // OK: ändrar innehållet
    constPtr  = nullptr;        // Fel! kan inte ändra pekaren

    *constPtrToConst = 0;       // Fel! kan inte ändra innehållet
    constPtrToConst  = nullptr; // Fel! kan inte ändra pekaren
}
```


Pekare

Pekare på konstanter och konstant pekare

```
int k;            // ändringsbar int
int const c = 100;// konstant int
int const * pc;   // pekare till konstant int
int *pi;          // pekare till ändringsbar int

pc = &c;   // OK
pc = &k;   // OK, men k kan inte ändras via *pc
pi = &c;   // FEL! pi får ej peka på en konstant
*pc = 0;   // FEL! pc pekar på en konstant

int* const cp = &k; // Konstant pekare
cp = nullptr;       // FEL! Pekaren ej flyttbar
*cp = 123;          // OK! Ändrar k till 123
```


`char[]`, `char*` och `const char*`
const är viktigt för C-strängar

En *string literal* (t ex `"I am a string literal"`) är **const**.

- Kan lagras i icke-skrivbart minne
- **char*** `str1 = "Hello";` — *deprecated* i C++ – ger varning
- **const char*** `str2 = "Hello";` — OK, strängen är **const**
- **char** `str3[] = "Hello";` — `str3` går att ändra

Typomvandlingar (casting)
Implicita Typomvandlingar

Automatiska typomvandlingar

- Uttryck av typen $x \odot y$, för någon binär operator $\odot$
  Ex: `double + int ==> double`
  `float + long + char ==> float`
- Tilldelningar och initieringar: Värdet i högerledet konverteras till samma datatyp som i vänsterledet
- Konvertering av typen hos aktuella parametrar till typen för de formella parametrarna
- Villkor i `if`-satser, etc. ⇒ **`bool`**
- C-array ⇒ pekare (*array decay*)
- `0` ⇒ `nullptr` (tom pekare, i C++11, tidigare definierade man ofta konstanten `NULL`)

Typomvandlingar (*casting*)
Explicita, namngivna typomvandlingar (C++-11)

- **`static_cast`**`<new_type> (expr)`
  - omvandlar mellan kompatibla typer (*kollar inte talområden*)
- **`reinterpret_cast`**`<new_type> (expr)`
  - inget skyddsnät, samma som C-stil
- **`const_cast`**`<new_type> (expr)` - lägger till eller tar bort **`const`**
- **`dynamic_cast`**`<new_type> (expr)` - används för pekare till klasser. Gör typkontroll vid *run-time*, som i Java.

Exempel

```
char c;              // 1 byte
int *p = (int*) &c;  // pekar på int: 4 bytes

*p = 5; //fel vid exekvering, stack-korruption

int *q = static_cast<int*> (&c); // kompileringsfel
```


Typomvandlingar (*casting*)
Explicita typomvandlingar, C-stil

Syntax

```
(typnamn)uttryck; // (Både i C och i C++, som i Java)
typnamn(uttryck); // (Alternativ syntax i C++)
```

För varianten `typnamn(uttryck)` måste `typnamn` vara *ett ord*, d v s `int *(...)` eller **`unsigned long`**`(...)` är inte OK.

- Stor risk att göra fel - använd namgivna typomvandligar
  - blir tydligare i koden, t ex **`const_cast`** kan bara ändra **`const`**
  - lätt att söka efter: casts är bland det första man tittar på när man letar fel
- Varning i GCC: `-Wold-style-casts`
- Vanlig i äldre kod

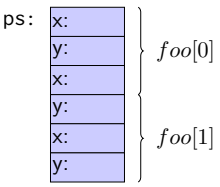
Typomvandlingar (*casting*)
Varnande exempel

```
struct Point{
    int x;
    int y;
};

Point ps[3];

struct Point3d{
     int x;
     int y;
     int z;
};

Point3d* foo = (Point3d*) ps;
```



Med *named casts* måste man använda **`reinterpret_cast`**`<Point3d*>`

med **`static_cast`** fås felet

```
invalid static_cast from type 'Point[3] to type 'Point3d*'
```


specialfall: **`void`**-pekare

En **`void`**`*` kan peka på vad som helst (objekt av godtycklig typ.)

I C omvandlas **`void`**`*` implicit till/från varje pekartyp.

I C++ omvandlas `T*` implicit till **`void`**`*`. Åt andra hållet krävs en explicit *type cast*.

Nästa föreläsning

- Felhantering, exceptions (5.6, 18.1.1)
- Dynamiskt minne och smarta pekare (12.1)
- `unique_ptr` (12.1.5)

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

const och constexpr 2.4

C-strängar (char*) 3.5.4–3.5.5

Typomvandlingar 4.11

Funktionsmallar 16.1.1

Pekare och textsträngar
C-strängar

Lagring i minnet

- *null-terminerad*: tecknet \0 markerar slutet på strängen
- `char namn[]= "Nils";  // längd = 5 bytes`

  är ekvivalent med

  `char namn[] = {'N', 'i', 'l', 's', '\0'};`
adress	data
namn	N
namn+1	i
namn+2	l
namn+3	s
namn+4	\0

Access av element

```
cout << namn[1] << namn[3];
```

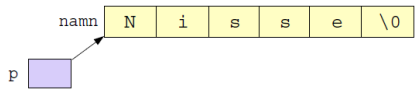
is

Pekare och textsträngar
C-strängar

Notera returtyperna

```
char namn[] = "Nisse";
char* p;
p = namn;
cout << p << endl;
cout << p+3 << endl;      // Skriver en delsträng  (typ: char*)
cout << *(p+3) << endl;   // Skriver ett tecken    (typ: char)
cout << namn + 3 << endl; // Samma som p+3
cout << namn[3] << endl;  // ett tecken
```

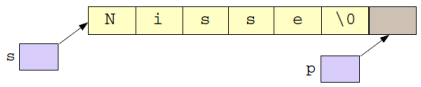
```
Nisse
se
s
se
s
```



Pekare och textsträngar

Puzzle Corner version av `strlen` (ger stränglängden)

```
// Användning av const char* för
// att garanterat inte ändra något
int strleng(const char *s) {
   const char *p; // Pekaren kan ändras!
   for (p = s; *p++; );
   return p-s-1;
}
```



Funktionspekare

Pekare kan också peka på funktioner

```
double hypotenuse(int a, int b) {
    return sqrt( a*a + b*b);
}

double mean(int x, int y) {
    return static_cast<double>(x+y)/2;
}

int main() {
    double (*pf)(int, int);

    pf = hypotenuse;
    cout << "hypotenuse: " << pf(3,4) << endl;

    pf = mean;
    cout << "mean: " << pf(3,4) << endl;
}
```


Funktionspekare

Funktionspekare kan vara argument till funktioner

```
float eval(float (*f)(int,int), int m, int n)
{
   return f(m, n);
}

float hypotenuse(int a, int b)
{
   return sqrt((float)(a*a + b*b));
}
float mean(int x, int y)
{
   return ((float)(x + y))/2;
}
int main ()
{
   cout << eval(hypotenuse, 3, 4) << endl;
   cout << eval(mean, 3, 4) << endl;
}
```

Funktionspekare
Alternativ i C++

Funktionspekare finns i C. I C++ finns även

- *funktor*: klass med **`operator()`**
- *lambda*: "anonym funktor"