EDAF30 – Programmering i C++

Generisk programmering med templates (mallar)

Sven Gestegård Robertz
Datavetenskap, LTH

2015

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& min(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 << min(x, y) << endl;
cout << min(m, n) << endl;

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

// För att forcera casting blir man explicit
cout << min<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 container

Variant som kan användas för en godtycklig *klass-template* som

- har elementtypen (T) som typparameter (kan ha fler: (...)
- har `begin()` och `end()`

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

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);
```


Klassmallar

- Containerklasserna `vector`, `deque` och `list` utgör exempel på klasser med *typparametrisering* eller *klassmallar*
- Kompilatorn genererar utgående från en sådan klassmall alla olika typer av klasser som behövs beroende på aktuell typ insatt som typparameter
- Slipper manuellt skriva en ny klass för varje enskild komponentdatatyp för att implementera en viss sammansatt datastruktur
- Klasser kan parametriseras
- Exempel: container-klasser i standardbiblioteket
  - `std::vector`
  - `std::deque`
  - `std::list`

Klassmallar
Container från f7

```
class Container {
public:
    virtual int size() const  =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};

class Vektor :public Container {
public:
    Vektor(int l = 10) :p{new int[l]},sz{l} {}
    ~Vektor() {delete[] p;}
    int size() const override {return sz;}
    int& operator[](int i) override {return p[i];}
    virtual void print() const override;
private:
    int *p;
    int sz;
};
```

- generalisera till godtycklig elementtyp

Klassmallar
Parametriserad Container och Vektor

```
template <typename T>
class Container {
public:
    virtual size_t size() const =0;
    virtual T& operator[](size_t o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};

template <typename T>
class Vektor :public Container<T> {
public:
    Vektor(size_t l = 10) :p{new T[l]},sz{l} {}
    ~Vektor() {delete[] p;}
    size_t size() const override {return sz;}
    T& operator[](size_t i) override {return p[i];}
    virtual void print() const override;
private:
    T *p;
    size_t sz;
};
```


Klassmallar
Definition av medlemsfunktioner

```
template <typename T>
void Vektor<T>::print() const
{
  for(size_t i = 0; i != sz; ++i)
    cout << p[i] << " ";
  cout << endl;
}
```

- Fungerar för alla typer som har **operator**<<
- men inte för element av typ

```
struct Foo{
   int x;

   Foo(int d=0) :x{d}{}
};
```

Specialisering av mallen för typen `Foo`:
(medlemmar i en klassmall är mallar)

```
template<>
void Vektor<Foo>::print() const
{
  for(size_t i = 0; i != sz; ++i)
    cout << "Foo("<<p[i].x << ") ";
  cout << endl;
}
```


Klassmallar
Exempel: binärt sökträd

```
template <class D>
class Tree {
public:
    Tree() : root(nullptr) {}
    Tree(D d) :root{new Node(d)}{}
    ~Tree() {delete root;}
    bool empty() const {return root == nullptr;}
    D& value() const {check(); return root->data;}
    Tree& l_child() const {check(); return root->left;}
    Tree& r_child() const {check(); return root->right;}
    void insert(D d);
    D* find(D d);
private:
    class Node {...}
    Node* root;
    void check() const {if(empty()) throw range_error("Empty tree");}
};
```

Node *som inre klass*

Klassmallar
Exempel: binärt sökträd

Nodklassen som inre klass

```
template <class D>
class Tree{
  //...}
  class Node{
      friend class Tree<D>;
      D data;
      Tree<D> left;
      Tree<D> right;
      Node(D d) : data{d} {}
      ~Node() {}
  };
};
```


Klassmallar
Exempel: binärt sökträd

Medlemsfunktioner i en klassmall är funktionsmallar

```
template <class D>
void Tree<D>::insert(D d)
{
    if(empty()){
        root = new Node(d);
    }else if( d<value()){
        l_child().insert(d);
    }else{
        r_child().insert(d);
    }
}
```

```
template <class D>
D* Tree<D>::find(D d)
{
    if(empty())
        return nullptr;
    else if(d==value()) {
        return &value();
    }else if(d < value()){
        return l_child().find(d);
    } else{
        return r_child().find(d);
    }
}
```


Klassmallar
Exempel: binärt sökträd

Användning:

Ett träd med heltal

```
void test_tree()
{
    Tree<int> t;

    t.insert(17);
    t.insert(11);
    t.insert(22);
    t.insert(19);

    try_find(t,22);
    try_find(t,19);
    try_find(t,23);
}
```

```
22 found
19 found
23 not found
```

Hjälpfunktion:

```
template <typename T>
void try_find(Tree<T>& t, const T& v)
{
    auto res = t.find(v);
    if(res) {
        cout << v << " found\n";
    }else {
        cout << v << " not found\n";
    }
}
```

Klassmallar
Exempel: binärt sökträd

Använda trädet för Person-objekt

```
struct Person{
    string pnr;
    string name;
    Person(string pn, string n) :pnr{pn},name{n} {}
};
Tree<Person> ps;

ps.insert(Person("121110-1516", "Kalle"));
```

```
|| /Users/sven/work1/kurser/cpp/test_templates.cpp:
In instantiation of 'void Tree<D>::insert(D)
[with D = Person]':

test_templates.cpp|699 col 45| required from here
test_templates.cpp|606 col 16 error| no match for
'operator<' (operand types are 'Person' and 'Person')
 ||      }else if( d<value()){
...
...
```


Klassmallar
Exempel: binärt sökträd

Lösning: använd "egenskapsklass" för att ge jämförelseoperationerna

Lägg till en typparameter till trädmallen

```
template <class D, class E>
class Tree {
public:
    using Comp = E;  // Lokalt namn för typparametern

    // samma som förut
private:
    class Node{
        friend class Tree<D>;
        D data;
        Tree<D,E> left;
        Tree<D,E> right;
        Node(D d) : data{d} {}
        ~Node() {}
    };
    // samma som förut
};
```


Klassmallar
Exempel: binärt sökträd med egenskapsklass

Medlemsfunktionerna använder komparator-klassen

```
template <class D, class E>
D* Tree<D,E>::find(const D& d)
{
  if(empty())
    return nullptr;
  else
  if(Comp::equal(d,value())) {    // eller E::equal
    return &value();
  }else
  if(E::less_than(d,value())){    // eller Comp::less_than
    return l_child().find(d);
  } else{
    return r_child().find(d);
  }
}
```


Klassmallar
Exempel: binärt sökträd med egenskapsklass

Comparator-klassmall för "vanliga" typer

```
template <typename T>
struct Comparator {
  static bool less_than(T l, T r) {return l < r;}
  static bool equal(T l, T r) {return l == r;}
};
```

Specialisering av Comparator för Person

```
template<>                  // specialisering -> ingen parameter
struct Comparator<Person>{ // Ange typen explicit här
    static bool less_than(Person l, Person r)
    {
        return l.pnr.compare(r.pnr) < 0;
    }
    static bool equal(Person l, Person r)
    {
        return l.pnr.compare(r.pnr) == 0;
    }
};
```


Klassmallar
Exempel: binärt sökträd

Lägg till en defaultparameter för Comparator

Kompilatorn väljer Comparator<D>

```
template <class D, class E=Comparator<D>>
class Tree {
  // samma som förut
};
```

Nu fungerar:

```
Tree<char> tc;
tc.insert('A');
tc.insert('c');
try_find(tc,'a');
try_find(tc,'c');

Tree<Person> ps;
ps.insert(Person("131211-1233", "Lisa"));
ps.insert(Person("010203-9876", "Nisse"));
try_find(ps, Person("010203-9876", "Vem"));
```


Mallar, kommentarer

- Mallar har parametrar
  - typ-parametrar: deklareras med **class** eller **typename**
  - värde-parametrar: deklareras som vanligt, t ex **int** N
- Kompilatorn behöver hela mallen för att kunna instansiera den
  ⇒ den ska ligga i *header-filen* (om andra ska inkludera den)
- Överlagring:
  - Funktioner kan övelagras ⇒ funktionsmallar kan överlagras
  - Klasser kan inte övelagras ⇒ klassmallar kan inte överlagras
- Specialisering:
  - Klassmallar kan specialiseras *partiellt* eller *fullständigt*
  - Funktionsmallar kan bara specialiseras *fullständigt*, *men*
    - Specialiseringar överlagras inte
    - Ofta bättre/tydligare att överlagra med en vanlig funktion (inte mall) än att specialisera

Vi har talat om

- parametriserade funktioner
- parametriserade klasser

Nästa föreläsning:

- "de sista pusselbitarna"
  - **namespace**
  - **union**
  - bit-operatorer
  - bit-fält
- Kommentarer och tips

Funktionsmallar
Funktionsmall för minsta elementet i en container

Variant för *godtycklig klass som har* `begin()` *och* `end()`

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

- *Trailing return type* (C++11)
- `decltype` : typoperator som ger *typen av ett uttryck* (`&`)
- om `Container` är `vector<int>` blir returtypen `int&`

Användning:

```
deque<int> d{5,3,1,2};
int md = min_element(d);

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