EDAF30 – Programmering i C++

10. Generisk programmering. Mallar och funktionsobjekt

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

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`

"Container" är ett generellt begrepp, oberoende av element-typen

Parametriserade typer

- Generalisera Vektor av doubles till Vektor av vad-som-helst.
- Klassmall med element-typ som mall-parameter

Exempel:

```
template <typename T>
class Vektor{
private:
    T* elem;
    int sz;
public:
    explicit Vektor(int s);
    ~Vektor() {delete[] elem;}

    // copy och move ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const {return sz;}
};
```


Klassmallen Vektor
Medlemsfunktioner

- Invariant:
  - sz $>= 0$ (*NB! deklarerad* **int** sz, *inte* **unsigned** sz)
  - elem pekar på en T[sz];

```
template <typename T>
Vektor<T>::Vektor(int s){
    if(s < 0) throw invalid_argument("Negative size");
    sz = s;
    elem = new T[sz];
};
template <typename T>
const T& Vektor<T>::operator[](int i) const
{
    if(i < 0 || size() <= i) throw range_error("Vektor::operator[]");
    return elem[i];
}
template <typename T>
T& Vektor<T>::operator[](int i)
{
    const auto& constme = *this;
    return const_cast<T&>(constme[i]);
}
```


Klassmallen Vektor
Konstructor med `std::initializer_list`

Vi vill kunna initera vektorer enkelt:

```
Vektor<int> vs{1,3,5,7,9};

template <typename T>
Vektor<T>::Vektor(initializer_list<T> l)
                  :Vektor<T>(static_cast<int>(l.size()))
{
    int pos{0};
    for(const auto& e : l){
        elem[pos++] = e;
    }
}
```

static_cast<int> *behövs eftersom*
`std::initializer_list<T>::size()` *returnerar en* **unsigned** *typ*

Mall-parametrar
Typer eller värden

```
template <typename T, int N>
struct Buffer{
    using value_type = T;
    constexpr int size() {return N;}
    T buf[N];
};
```

- `Buffer`: som en array, som känner till sin storlek
  - Inget overhead för heap-allokering
  - mall-parametrar måste vara **constexpr**
    → kan inte ha variabel storlek
  - jfr `std::array`
- Storleken som värde-parameter till mallen
- Ett alias (`value_type`) och en **constexpr**-funktion (`size()`)
  - Användare kan komma åt (läsa) mall-parametrarna

Mall-parametrar
och alias

Alla standard-containers har alias `value_type`

```
template <typename T>
class Vektor{
public:
    using value_type = T;
    ...
};

template <typename T>
typename T::value_type get_first(T& t)
{
    return t[0];
}

void example()
{
    Vektor<int> v{2,4,3,5,4,6};
    cout << "first element of v is " << get_first(v) << endl;
}
```

Här behöver **typename** *anges för att kompilatorn ska veta att namnet* `value_type` *är en typ innan den instantierat mallen*

Alias
Binda parametrar

Man kan använda **using** för att skapa alias för typer

```
using size_t = unsigned int;
```

inklusive mallar

```
using IntVektor = Vektor<int>;
```


Klassmallar
Container från f8

```
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:
    explicit Vektor(int l);
    ~Vektor();
    int size() const override;
    int& operator[](int i) override;
    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:
    using value_type = T;
    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 = 0) :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;
}
```

- Medlemsfunktioner i en klassmall är funktionsmallar
- `print()` fungerar för alla typer som har **operator**<<
- *"Duck typing":*
  if it walks like a duck and quacks like a duck, it is a duck

Kompilering av mallar

- Typkontrollen vid kompilering av en mall tittar på *användningen* av argumenten
  - användningen, i *definitionen* av mallen
  - . . . i stället för mot en typ i *deklarationen* (som för vanliga funktioner)
- skillnad mot den "vanliga" varianten, där typen (klassen) styr vad en operation betyder
- görs vid kompilering i C++
  - Kompilatorn måste se *definitionen* av mallen där den används
  - Hela mallen måste ligga *i header-filen*
- Jämför med dynamiska språk som Python

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;
}
```


Specialsering av mallar

- Klassmallar kan specialiseras
  - fullständigt
  - partiellt
- Funktionsmallar kan specialiseras
  - fullständigt
  - *men överlagring är alltid att föredra*

Funktionsobjekt
Exempel

```
template<typename T>
class Less_than {
    const T val;
public:
    Less_than(const T& v) :val{v} {}
    bool operator()(const T& x) {return x < val;}
};

void use_less_than()
{
    Less_than<int> lt5{5};
    Vektor<int> v{1,7,6,2,8};

    for(auto x : v) {
        cout << x << " < 5:" << boolalpha << lt5(x) << endl;
    }

}
```


Funktionsobjekt
Exempel

Med en funktionsmall:

```
template<typename C, typename P>
int count(const C& c, P& pred) {
        int res{0};
        for(const auto& x : c) {
            if(pred(x)) ++res;
        }
        return res;
}
```

kan vi använda vårt funktionsobjekt:

```
void use_functors()
{
    Less_than<int> lt5{5};
    Vektor<int> v{1,7,6,2,8};

    cout << "count numbers < 5: " << count(v,lt5) << endl;

}
```


Iteratorer

För att kunna använda vår `Vektor<T>` med *range-for* och standard-algoritmer behövs funktionerna `begin()` och `end()`

```
template <typename T>
const T* begin(const Vektor<T> &v)

template <typename T>
T* begin(Vektor<T> &v)

template <typename T>
const T* end(const Vektor<T>& v)

template <typename T>
T* end(Vektor<T>& v)
```

Iteratorer
const-versionerna

För att kunna använda vår `Vektor<T>` med *range-for* och standard-algoritmer behövs funktionerna `begin()` och `end()`

Const-versionerna

```
template <typename T>
const T* begin(const Vektor<T> &v)
{
  return v.size() ? &v[0] : nullptr;
}

template <typename T>
const T* end(const Vektor<T>& v)
{
  return begin(v)+v.size();
}
```


Iteratorer
icke-const-versionerna

- Undvik kod-duplicering
  - Använd **const**-versionerna
  - Användningsexempel för **const_cast**

icke-const-versionerna

```
template <typename T>
T* begin(Vektor<T> &v)
{
  return const_cast<T*>(begin(const_cast<const Vektor<T>&>(v)));
}

template <typename T>
T* end(Vektor<T>& v)
{
  return const_cast<T*>(end(const_cast<const Vektor<T>&>(v)));
}
```


Förklaring
icke-const-versionerna

```
template <typename T>
T* begin(Container<T> &v)
{
    return const_cast<T*>(begin(const_cast<const Container<T>&>(v)));
}
```

kan även skrivas

```
template <typename T>
T* begin(Container<T> &v)
{
    const Container<T>& cv= v;     skapa const-referens
    const T* cbegin = begin(cv);   returvärdet är const-pekare
    return const_cast<T*>(cbegin);         gör om till icke-const-pekare
}
```

*NB! Anropar **const**-versionen från icke-**const**-versionen. Aldrig åt andra hållet.*

Variadic templates
En funktionsmall kan ha variabelt antal argument

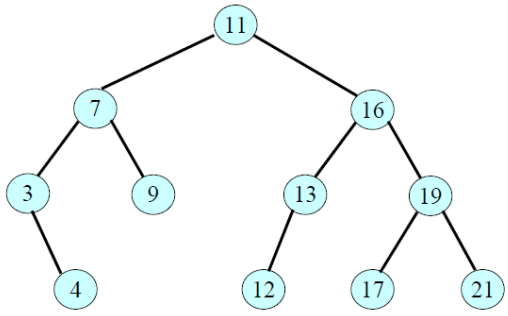
```
void println() { basfall: inga argument
    cout << endl;
}

template <typename T, typename... Tail>
void println(T head, Tail... tail)
{
    cout << head << " ";     Skriv första elementet
    println(tail...);        rekursion: skriv ut resten
}

void test_variadic()
{
    string a{"Hej"};
    int b{10};
    double c{17.42};
    long d{100};

    println(a,b,c,d);

}
```


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 överlagras ⇒ funktionsmallar kan överlagras
  - Klasser kan inte överlagras ⇒ 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

Träd

Binärt sökträd: Alla noder i vänstra delträdet har mindre nyckelvärden än roten och alla noder i högra har större värden



Träd

Egenskap hos binära sökträd: Utskrift i *inorder* ger värdena i *växande ordning* (sorterat)

3 4 7 9 11 12 13 16 17 19 21

Sökning efter visst värde

```
Nod* find(Nod* root, int value) {
    if (root == nullptr)
        return nullptr;
    else if (value == root->data)
        return root;
    else if (value < root->data)
        return find(root->left, value);
    else
        return find(root->right, value);
}
```


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"));
```

```
|| 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"));
```


Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Generisk programmering 16.1, 16.3 (16.4–16.5)