Programmering i C++
EDA623
Mallar

Mallar

Innehåll

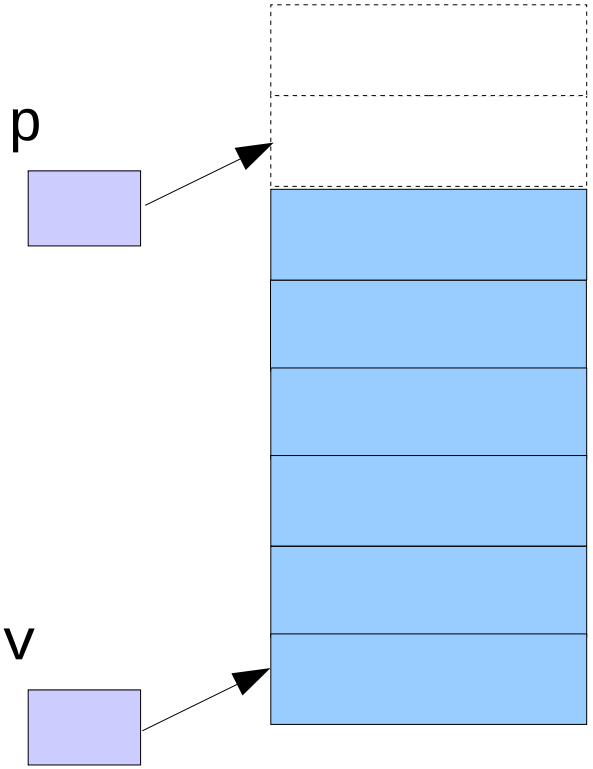
- Klassmallar
- Funktionsmallar

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

Exempel: Egen generisk stackklass

```
template <class T>
class Stack {
    T *v, *p;
    int sz;
public:
    Stack(int s) {v = p = new T[sz=s];}
    ~Stack() {delete [] v;}
    void push(T a) {*p++ = a;}
    T pop() {return *--p;}
    int size() const {return p-v;}
};
```



`T` utgör här en s.k. *typparameter*

Klassmallar

En stack för upp till 100 st `char` kan skapas så här:

```
Stack <char> sc(100);
```

vilket ger samma resultat som

```
class Stack_char {
    char *v, *p;
    int sz;
public:
    Stack_char(int s) {v = p = new char[sz=s];}
    ~Stack_char() {delete [] v;}
    void push(char a) {*p++ = a;}
    char pop() {return *--p;}
    int size() const {return p-v;}
};

// ...
Stack_char sc(100);
```

Klassmallar

Annan variant av klassmall (utan `inline`)

```
template <class T>
class Stack {
public:
    Stack(int s);
    ~Stack();
    void push(T a);
    T pop();
    int size() const;
private:
    T *v, *p;
    int sz;
};
```

Klassmallar

Annan variant av klassmall (utan `inline`)

```
// Medlemsfunktioner def. i .h-filen utanför klassen
template<class T> void Stack<T>::push(T a) {
    *p++ = a;
}
template<class T> T Stack<T>::pop() {
    return *--p;
}
template<class T> Stack<T>::Stack(int s) {
    v = p = new T[sz = s];
}

template<class T> Stack<T>::~Stack() {
    delete []v;
}
```

Exempel på användning av Stack

```
Stack<double> sd(200);

void f(Stack<Komplex>& sk) {
    sk.push(Komplex(2.1, 3.44));
    Komplex z = 0.45*sk.pop();
    sk.push(z);
    Stack<int> *p = 0;
    p = new Stack<int>(200);
    for (int i = 0; i<200; i++) {
        p->push(i);
        sd.push(3.14*i);
    }
}
```

Klassmallar

Även s.k. *värdeparametrar* kan användas

```
template <class T, int siz>
class Stack {
public:
    Stack() {v = p = new T[sz=siz];}
    ~Stack() {delete [] v;}
    void push(T a) {*p++ = a;}
    T pop() {return *--p;}
    int size() const {return p-v;}
private:
    T *v, *p;
    int sz;
};
//
// p = new Stack<int>(200); // ersätts med
// p = new Stack<int, 200>;
```

Statiska medlemmar

```
template <class T>
class Stack {
    //...
    static int nst = 0;
public:
    //...
    static int antal_stackar() {return nst;}
};
```

Varje typ av stack (`Stack<int>`, `Stack<char>` osv) har en egen upplaga av klassvariabeln `nst`

Klassmallar

En trädklassmall ...

```
template<class D>
class Trad {
public:
    Trad() : rot(nullptr) {}
    Trad(D d) { rot = new Nod<D>(d); }
    ~Trad() { delete rot; }
    bool tomt() const { return rot == nullptr; }
    D& varde() const { koll(); return rot->data; }
    Trad& v_barn() const { koll(); return *rot->vanster; }
    Trad& h_barn() const { koll(); return *rot->hoger; }
    void satt_in(D d);
    D* sok(D d);
private:
    // class Nod {};
    Nod *rot;
    void koll() const {if (tomt()) throw range_error("Trad");}
};
```

Klassmallar

... med den inre nodklassen

```
template<class D>
class Trad {
public:
// ...
private:
    class Nod {
        friend class Trad<D>;
        D data;
        Trad<D> vanster, hoger;
        Nod(D d) : data(d) {}
        ~Nod() {}
    };
// ...
};
```

Medlemsfunktioner i trädklassmallen

```
// Insättning i binärt sökträd
template<class D>
void Trad<D>::satt_in(D d) const {
    if (tomt())
        rot = new Nod<D>(d);
    else if (d < varde())
        v_barn().satt_in(d);
    else
        h_barn().satt_in(d);
}
```

Medlemsfunktioner i trädklassmallen

```
// Sökning i binärt sökträd
template<class D>
D* Trad<D>::sok(D d) {
    if (tomt())
        return nullptr;
    else if (d == varde())
        return &varde();
    else if (d < varde())
        return v_barn().sok(d);
    else
        return h_barn().sok(d);
}
```

Klassmallar

Skapande av ett träd med heltal

```
//...

Trad<int> t;

cin >> i;
t.satt_in(i);
cin >> i;
if (t.sok(i))
    cout << "finns i trädet" << endl;
```

Klassmallar

Skapande av ett träd med personer istället heltal

```
class Person {
public:
    char pnr[11];
    char namn[40];
};

//...
Trad<Person> t;

// ... motsvarande operationer som innan
// men denna gång fungerar det inte
// eftersom jämförelse inte är definierat
// för Person-klassen
```

Klassmallar

Trick: Använd en s.k. *egenskapsklass* (*traits class*) med statiska funktioner `mindre` och `lika` vilka ersätter < resp. ==.

```
// Modifiering av klassen Trad med
// egenskapsklass E
template<class D, class E>
class Trad {
    // Samma som innan
};
```

Modifiering av medlemsfunktionerna

```
template<class D, class E>
void Trad<D,E>::satt_in(D d) {
    if (tomt())
        rot = new Nod<D>(d);
    else if (E::mindre(d, varde()))
        v_barn().satt_in(d);
    else
        h_barn().satt_in(d);
}
```

Modifiering av medlemsfunktionerna

```
template<class D, class E>
D* Trad<D>::sok(D d) {
    if (tomt())
        return 0;
    else if (E::lika(d, varde()))
        return &varde();
    else if (E::mindre(d, varde()))
        return v_barn().sok(d);
    else
        return h_barn().sok(d);
}
```

Klassmallar

För att det fortfarande ska fungera för de fall där < och == är definierade kan följande användas

```
template<class D>
class TradE {
public:
    static bool lika (const D& d1, const D& d2)
        {return d1 == d2;}
    static bool mindre(const D& d1, const D& d2)
        {return d1 < d2;}
}

// Skapa ett träd
Trad<int, TradE<int> > t;
```

Klassmallar

För att det ska fungera för t.ex. klassen `Person` måste en *specialinstans* av klassmallen användas

```
class TradE<Person> {
public:
    static bool lika (const D& d1, const D& d2)
        {return strcmp(d1.pnr, d2.pnr) == 0;}
    static bool mindre(const D& d1, const D& d2)
        {return strcmp(d1.pnr, d2.pnr) < 0;}
}

// Skapa ett träd
Trad<Person, TradE<Person> > t;
```

OBS: En specialinstans av en klassmall skall *inte* föregås av `template`

Funktionsmallar

Även vanliga funktioner kan vara generiska dvs ha mallar, s.k. *funktionsmallar*

Exempel: Vanliga minimifunktionen

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

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;
```

Funktionsmallar

Funktionsmall för minsta elementet i ett fält

```
template<class T>
T& min_element(T f[], int n) {
    int m = 0;
    for (int i = 1; i<n; i++)
    if (f[i] < f[m])
        m = i;
    return f[m];
}
```

Funktionsmallar

Funktionsmallar för byte och sortering

```
template<class T>
void byt(T& a, T& b) {
    T temp = a; a = b; b = temp;
}

template<class T>
void sort(T f[], int n) {
    for (int k = 0; k<n; k++)
        byt(f[k], min_element(f + k, n - k));
}
```

Funktionsmallar

Standardalgoritmerna är funktionsmallar!

```
// Typparametern IT nedan är en iterator el. pekartyp
template<class IT>
IT min_element(IT first, IT last) {
    IT m = first;
    for (IT i = ++first; i != last; i++)
    if (*i < *m)
        m = i;
    return m;
}

template<class IT>
void sort(IT first, IT last) {
    for (IT k = first; k != last; k++)
        byt(*k, *min_element(k, last));
}
```

Funktionsmallar

Exempel på användning av algoritmerna

```
double a[11] = { 11, 7, 3, 4, 9, 16, 13, 12, 19, 17, 21 };
list<int> l = { 11, 7, 3, 4, 9, 16, 13, 12, 19, 17, 21 };
//...
cout << *min_element(a, a + 11) << endl;
cout << *min_element(l.begin(), l.end()) << endl;
sort(a, a + 11);
//skriv(a, a + 11);
sort(l.begin(), l.end());
//skriv(l.begin(), l.end());
```

Funktionsmallar

För klasser som inte har operatorn < definierad:

```
// Compare är en egenskapsklass av funktionsobjektstyp
template<class IT, class Compare>
IT min_element(IT first, IT last, Compare cmp) {
    IT m = first;
    for (IT i = ++first; i != last; i++)
    if (cmp(*i, *m))
        m = i;
    return m;
}
// Egenskapsklassen måste ha funktionsdefinitionsop.:
class Str_Less_Than {
public:
    bool operator () (char *s1, char *s2)
    {
        return strcmp(s1, s2) < 0;
    }
};
```

Funktionsmallar

Exempel på användning med stränglista:

```
//
list<char *> tl = { "hej", "du", "glade" };
Str_Less_Than lt; // funktionsobjekt
// ...
cout << *min_element(tl.begin(), tl.end(), lt);
// Annan variant av ovanstående anrop:
cout << *min_element(tl.begin(), tl.end(), Str_Less_Than());
// Här skapas automatiskt ett objekt av typ Str_Less_Than
```