Introduktion till templates

7 INTRODUKTION TILL TEMPLATES

TYPPARAMETERISERING (TEMPLATES)

Templates

Template för funktioner

Definition av templatefunktion
Exempel med template-funktion
Mer exempel med templatefunktioner

Klass som template

Deklaration av templateklass
Definition av klassmedlemmar i templateklass
Definiera specialfall av template
Använda templateklass

Övningar

Övning 1
Övning 2

Typparameterisering (Templates)

Templates

C++-programmeraren konstruerar ibland att funktioner och klasser som liknar varandra. Detta är ofta fallet när man överlagrat en funktion och denna finns i versioner som skiljer sig åt endast i avseende på argumentens typ men är identiska i avseende på funktionalitet och algoritmer.

Med hjälp av **templates** behöver programmeraren endast skriva **en** sådan funktion. Denna får sedan fungera som mall (template). Kompilatorn genererar sedan en version för varje typ av anrop eller explicit deklaration som programmeraren gjort.

Templates (eller typparameterisering) är ett tillägg till C++ som finns i senare implementationer (2.1-) av C++. Det är därför inte säkert att din C++-kompilator kan använda templates.

Template för funktioner

template är ett reserverat ord i C++. Med hjälp av template kan man implicit använda en **typ som parameter**.

Definition av templatefunktion

För att definiera en funktion som template gör man på följande sätt:

```
template <class Type>
Type max(Type a, Type b)
{
     Type max;
     if (a>=b)
          max = a;
     else
          max = b;
     return max;
}
```

I ovanstående templatedefinition är **Type** en variabel parametertyp som kompilatorn substituerar i vår mall för att kunna generera en ny variant på vår funktion.

När vi anropar funktionen med en viss typ av parameter räknar kompilatorn ut att en matchande funktion måste genereras enligt mallen. Om man t.ex. gör nedanstående anrop kommer kompilatorn att generera en **max**-funktion som tar två double och sedan en som tar två int.

```
     cout  << "max(56.5,56.7)       = " << max(56.5,56.7) << endl;
     cout  << "max(10,20)           = " << max(10,20) << endl;
```

Dvs **kompilatorn** genererar funktioner med ungefär nedanstående utseende:

```
double max(double a, double b)
{
     double max;
     if (a>=b)
          max = a;
     else
          max = b;
     return max;
}
int max(int a, int b)
{
     int max;
     if (a>=b)
          max = a;
     else
          max = b;
     return max;
}
```

Om vi däremot anropar på nedanstående sätt kommer kompilatorn att generera en variant som tar två pekare och jämför dessa efter storlek på adress vilket inte blir så lyckat (vi vill ha en lexikalisk jämförelse).

```
     cout  << "max(\"Urban\",\"Nisse\") = "
                                << max("Urban","Nisse") << endl;
```

Vi kan då definiera ett specialfall av funktionen max för parametrar av typen **char*.**

```
// Specialfall.....
char *max(char *a, char *b)
{
     if (strcmp(a,b)>0)
          return a;
     else
          return b;
}
```

Exempel med template-funktion

```
// templa1.cpp
#include <iostream.h>
#include <string.h>
template <class Type>
Type max(Type a, Type b)
{
     Type max;
     if (a>=b)
          max = a;
     else
          max = b;
     return max;
}
// Specialfall.....
char *max(char *a, char *b)
{
     if (strcmp(a,b)>0)
          return a;
     else
          return b;
}

int
main(int,char**)
{

     cout  << "max(56.5,56.7)       = " << max(56.5,56.7) << endl;
     cout  << "max(10,20)           = " << max(10,20) << endl;
     cout  << "max(\"Urban\",\"Nisse\") = "
                                << max("Urban","Nisse") << endl;
     return 0;
}
```

Mer exempel med templatefunktioner

Sorteringsalgoritmer är oftast generella och oberoende av datatyp. Därför passar det bra att använda en template där vi definierar algoritmen som sedan kompilatorn kan använda för att instansiera samma algoritm för olika typer.

```
// templa2.cpp
#include <iostream.h>
#include <string.h>

// En mall för att byta plats på två objekt av godtycklig typ..
template <class Type>
void swap(Type& i,Type& j)  {
       Type temp = i;
       i = j;
       j = temp;
}

// En mall för att sortera objekt av godtycklig typ..
template <class Type>
void sortera( Type faelt[], unsigned antal)
{
         for(register unsigned  i=0;i<antal-1;i++)
       for( register int j=i+1;j<antal;j++)
         if ( faelt[i] > faelt[j])
              swap(faelt[i],faelt[j]);
}

// En mall för att skriva ut objekt av godtycklig typ..
template <class Type>
void skrivut(Type faelt[], unsigned antal)
{
       for(register unsigned i=0;i<antal;i++) {
              cout.width(8);
              cout << faelt[i];
       }
       cout << endl;
}

main()
{
       int     ifield[]  = { -1 , -50 , 400 , 233 , 16 };
       double  dfield[]  = { 5.5 , 6.3 , 1.2 , -4.50 , 5.98 };

       sortera(ifield,sizeof(ifield)/sizeof(ifield[0]));
       sortera(dfield,sizeof(dfield)/sizeof(dfield[0]));

       skrivut(ifield,sizeof(ifield)/sizeof(ifield[0]));
       skrivut(dfield,sizeof(dfield) /sizeof(dfield[0]));

       return 0;
}
```

För att sedan använda en template för en funktion räcker det att man anropar funktionen. Kompilatorn genererar funktioner som matchar anropen (enligt den mall (template) som angetts):

Kompilatorn genererar kod som matchar anropen. Som argument kan man även använda egendefinierade typer av objekt (klasser) om dessa typer klarar att hanteras enligt algoritmen i templatefunktionen.

Klass som template

Det är också möjligt att deklarera en **templateklass**. Denna kan sedan vara mall för en viss typ av klasser. Så här deklareras en **templateklass**:

Deklaration av templateklass

```
template <class Type>
class Value {
    friend ostream& operator<< (ostream& os,const Value<Type>&);
    friend istream& operator>> (istream& is,Value<Type>&);
public:
    Value();
    Value(Type init);
    void print();
    operator Type();
private:
    Type value;
};
```

Definition av klassmedlemmar i templateklass

Nedan visar vi hur medlemmarna skall definieras om vi gör det utanför klassdefinitionen:

```
template <class Type>
Value<Type>::Value()
:value(0)
{
}
template <class Type>
Value<Type>::Value(Type init)
:value(init)
{
}
template <class Type>
Value<Type>::operator Type()
{
    return value;
}
template <class Type>
ostream& operator<<(ostream& os,const Value<Type>& v)
{
    return os << v.value;
}
template <class Type>
istream& operator>>(istream& is,Value<Type>& v)
{
    return is >> v.value;
}
```

Definiera specialfall av template

Man kan definiera specialfall av en template. Antag att vi skulle vilja använda en **char*** som parameter till template och i så fall hantera detta fall separat.

```
class Value <char*> {
    friend ostream& operator<<(ostream& os,const Value<char*>&);
    friend istream& operator>>(istream& is,Value<char*>&);
public:
    Value<char*>();
    Value<char*>(char* init);
    ~Value();
    void print();
    operator char*();
private:
    char* value;
};

Value<char*>::Value()
:value(0)
{
}
Value<char*>::Value(char* init)
:value(strcpy(new char[strlen(init)+1],init))
{
}

Value<char*>::~Value()
{
    delete [] value;
}

Value<char*>::operator char*()
{
    return value;
}

ostream& operator<<(ostream& os,const Value<char*>& v)
{
    return os << v.value;
}

istream& operator>>(istream& is,Value<char*>& v)
{
    return is >> v.value;
}
```

Använda templateklass

För att använda ovanstående templateklass gör man på nedanstående sätt, dvs "anropar" templateklassen med en typ som parameter (t.ex. int). Vi kan t.ex. använda vår mall när vi definerar typ på returvärden från en funktion, typ för formella parametrar samt lokala ellr globala variabler:

```
Value<int> funk(Value<int> a)
{
     Value <int> resultat;
     resultat = a * a;
     return resultat;
}
Value<double> funk(Value<double> d)
{
     Value <double> resultat;
     resultat = d * d;
     return resultat;
}
Value<int> funk(Value<char*> string)
{
     return strlen(string);
}
template <class Type>
Type max(Type a, Type b)
{
     return a > b ? a : b;
}
```

Nu kan vi anropa vår templateklass och våra templatefunktioner:

```
main()
{
     Value <int> a = 10;
     Value <int> b = 20;
     Value <int> c = a + b;

     cout << c <<  endl;
     cout << funk(c) << endl;
     cout << max(a,b) << endl;


     Value <double> d = 15.4;
     Value <double> e = 20.8;
     Value <double> f = d + e;

     cout << f <<  endl;
     cout << funk(f) << endl;
     cout << max(d,e) << endl;

     Value <char*> s1 = "Hej ";
     Value <char*> s2 = "Alla ";
     Value <char*> s3 = "Programmerare ";

     cout << s1 << s2 << s3 << endl;

     cout << funk(s1) << endl;
     cout << funk(s2) << endl;
     cout << funk(s3) << endl;

     return 0;
}
```

Genom att vi gör anropet "Value <int> a" genereras nedanstående klass samt motsvarande medlemsfunktioner:

```
class Value {
      friend ostream&
            operator<<(ostream& os, const Value<int>&);
      friend istream& operator>> (istream& is,Value<int>&);
public:
      Value();
      Value(int init);
      void print();
      operator int ();
private:
      int value;
};
Value<int>::Value()
:value(0)
{
}

Value<int>::Value(int init)
:value(init)
{
}
Value<int>::operator int()
{
      return value;
}
```

Genom att vi gör anropet "**Value <double> d;**" genereras nedanstående klass samt motsvarande medlemsfunktioner:

```
class Value {
      friend ostream&
            operator<<(ostream& os, const Value<double>&);
      friend istream& operator>> (istream& is,Value<double>&);
public:
      Value();
      Value(double init);
      void print();
      operator double ();
private:
      double value;
};
Value<double>::Value()
:value(0)
{
}

Value<double>::Value(double init)
:value(init)
{
}
Value<double>::operator double()
{
      return value;
}
```

Genom att vi gör anropen "**cout << c << endl;**","**cout << c << endl;**" och "**cout << s1 << endl;**" genereras nedanstående funktoner:

```
ostream& operator<<(ostream& os,const Value<double>& v)
{
      return os << v.value;
}

ostream& operator<<(ostream& os,const Value<int>& v)
{
      return os << v.value;
}
ostream& operator<<(ostream& os,const Value<char*>& v)
{
      return os << v.value;
}
```

Genom att vi gör nedanstående anrop:

```
...
cout << max(a,b) << endl;
...
cout << max(d,e) << endl;
...
```

Genereras nedanstående funktioner:

```
Value<int> max(Value<int> a,Value<int> b)
{
    return a > b ? a : b;
}

Value<double> max(Value<double> a,Value<double> b)
{
    return a > b ? a : b;
}
```

Övningar

Övning 1

Gör **två** templatefunktioner. En som beräknar summan för ett godtyckligt antal element i ett fält med element av godtycklig enkel typ samt en som beräknar medelvärdet för elementen. Eventuellt kan det passa att anropa den ena funktionen från den andra.

Övning 2

Gör en template för en listklass för lagring av element av godtycklig typ. Man skall kunna lägga in nya objekt, iterera över alla objekt samt flusha (dvs tömma listan).