Programmieren:

Templates

Inhaltsverzeichnis:

1. Verwendung

Zum Beispiel bei einer Containerklasse. Containerklassen werden **oft benötigt**. Sie beinhalten `char, int, Student` und viele weitere Möglichkeiten. Anstatt den selben Quellcode, in dem `char` zu `int` usw. geändert wird, immer wieder von neuem zu schreiben, ist es einfacher, ein **Template** zu erstellen und jeweils anzugeben, für welchen Datentyp diese Implementierung gelten soll.

2. Erstellen einer Templateklasse

Im Quellcode wird der entsprechende Datentyp, der „variabel" werden sollte, durch das Schlüsselwort `Type` im Header-File und im Source-File(.cpp) ersetzt (könnte auch ein anderer unverkennbarer Name sein, z.B. `T`). Im Header sollte vor der Klassendeklaration der Eintrag `template <class Type>` enthalten sein (ohne `;` am Schluss).

```
#ifndef STACKT_H
#define STACKT_H

template <class Type>    //ist Deklaration eines Templates
class Stack {
public:
     Stack( int size = defaultsize );      //Konstruktor
     Stack( const Stack<Type>& einStack ); //Kopierkonstruktor
     ~Stack();                             //Destruktor
     void push( Type c);
     Type pop();
     int size() const;
private:
     Type* data;
     const int maxLength;
};
#endif
```

`Type` kann durch einen beliebigen Datentyp (auch Klassen) ersetzt werden. Besondere Beachtung ist auch dem **Kopierkonstruktor** zu schenken mit der Silbe `const Stack<Type>&`. Bei der Angabe der Länge (`size`) des Stacks wird für jedes Datenformat ein Ganzzahlwert gebraucht. Diese Variable ist daher nicht abhängig davon, welches Datenformat / welche Klasse die Klasse `Stack` beinhaltet. Darum bleibt der Datentyp `int`.

In der Definition muss vor jeder Methode `template <class Type>` vorangesetzt werden, damit der Compiler `Type` als Parameter erkennt. Ausserdem ist der Name der parametrisierten Klasse `Stack<Type>`, daher ist dies vor dem Sichtbarkeitsoperator **`::`** anzugeben.

```
#include „stack.h“

template <class Type>
Stack<Type>::Stack( int size )     //Konstruktor
     : max length( size ), topindex( emptyIndex )
{
     data = new Type[size];
}

template <class Type>
Stack<Type>::~Stack()         //Destruktor
{
     delete [] data;
}
```

In `main()` kann dann folgendermassen auf die Template-Klasse zugegriffen werden:

```
void main()
{
     int i = 0;
     char einText[] = „Das ist ein Text“;
     Stack<char> charStack1;

     charStack1.reset(); //Ruft reset-Funktion von Stack auf
     Stack<float> floatStack1;
}
```

Die Verwendung der Objekte erfolgt wie bei normalen Objekten: die Methoden werden normal benutzt, wobei Parametertypen und Rückgabetypen nun ebenfalls statt `Type` z.B. `char` enthalten.

3. Faustregel zum Erstellen eines Klassentemplates

- Normale Klasse mit irgendeinem Datentyp schreiben, der später durch den Templateparameter ersetzt werden soll und Klasse gründlich testen
- Datentyp durch `Type` ersetzen
- Der Klassendeklaration und jeder Methodendefinition `template <class Type>` voranstellen
- Klassenname durch `Klassenname<Type>` ersetzen: überall, ausser beim Namen der Konstruktoren und des Destruktors

  Also beim Kopierkonstruktor:
  `Klassenname( const Klassenname<Type>& X);`
  Vor Klassenname **kein** Type, jedoch in Argumentliste, da ich ja ein Type-Objekt kopieren will.

4. Friend

Klassentemplates können auch `friend`s enthalten. Es sind hier zwei Fälle zu unterscheiden:

- Der Friend enthält den Typparameter nicht:
  - d.h. eine Funktion oder eine Klasse wird als `friend` deklariert
  - sie ist unabhängig vom Typparameter,

  dann ist die Funktion oder die Klasse `friend` von allen Instantiierungen des Templates.
- der Friend enthält den Typparameter:
  - d.h. eine Funktion oder eine Klasse wird als `friend` deklariert
  - sie ist abhängig vom Typparameter, d.h.
    - die `friend`-Funktion hat Parameter des Typs oder den Typ als Rückgabetyp
    - die `friend`-Klasse ist mit diesem Typ ebenfalls parametrisiert,

  dann ist die Funktion oder die Klasse nur `friend` einer Instatiierung des Templates.

Beispiel:

```
template <class Type>
class VektorIterator;    //Vorwärtsdeklaration VektorIterator

template <class Type>
class Vektor {
public:
...
private:
     friend VektorIterator<Type>
};

template <class Type>
Vektor<Type> operator*( Type x, const Vektor<Type>& v );
//geht auch bei Funktion

template <class Type>
class VektorIterator {
public:
     ...
};
#endif
```

5. Deklaration Templateklasse

Was muss bei der Umschreibung einer Klasse in ein Template in der Deklaration geändert werden?

- den Klassendeklarationen wurde `template <class Type>` vorangestellt
- der variable (zu ändernde) Datentyp wurde überall in `Type` geändert
- die Klassennamen wurden in `Vektor<Type>` bzw. `VektorIterator<Type>` geändert, ausser bei `class Vektor` und bei den Namen von Konstruktoren und Destruktoren
- `VektorIterator` wird `friend` von `Vektor`

```
template <class Type>
class VektorIterator;          //Vorwärtsdeklaration

template <class Type>
class Vektor {
public:
     Vektor( int n = 10 );     //Konstruktor (ohne <Type>)
     Vektor( const Vektor<Type>& v );   //Copy-Konstruktor
     Type& operator[]( int i );
     void print( ostream& out) const;
     ...
private:
     friend VektorItearator<Type>;
int length;
Type* p;
};

template <class Type>
Vektor<Type> operator*( Type x, const Vektor<Type>& v );
...

template <class Type>
class VektorIterator {
public:
     VektorIterator( Vektor<Type>& v );
     ...
private:
     Vektor<Type>& vr;
};
```

6. Definition Templateklasse

Was muss bei der Umschreibung einer Klasse in ein Template in der Definition geändert werden?

- den Methodendefinitionen wurde `template <class Type>` vorangestellt
- der variable (zu ändernde) Datentyp wurde überall in `Type` geändert
- die Klassennamen wurden in `Vektor<Type>` bzw. `VektorIterator<Type>` geändert, ausser bei den Namen von Konstruktoren und Destruktoren

```
#include „vektor.h“          //Templateklasse includen

template <class Type>
VektorIterator<Type>::VektorIterator(Vektor<Type>& v)
     : vr( v ), index( 0 )
{}

template yclass Type>
void
VektorIterator<Type>::first()
{
     index = 0;
}
...
```

7. Benutzung einer Templateklasse in main()

In folgenden Beispiel wird ein `Vektor` von `float` instantiiert, zur Iteration wird ein `VektorIterator` mit `float` instantiiert.

```
void main()
{
     float feld[12] = 1.1, ... };
Vektor<float> v( feld, 5);    //Instantiierung mit float
cout << v;

for( VektorIterator<float> vi( v); !vi.done(); vi.next() ) {
     vi.current() = 111.1;
}
cout << v;
```

8. Instantiierung

Eine Template-Definition führt alleine noch nicht zu lauffähigem Programmcode. Der Compiler generiert erst für jeden konkreten Parametersatz (z.B. float, int etc.) eine **spezifische Instanz** der Template-Funktion. Das heisst, dass er die Template-Klasse erst dann durch den gewählten Datentyp generiert.

Eine Instanzierung kann auf drei Arten erfolgen:

1. Beim erstmaligen Aufruf der Funktion leitet der Compiler aus den Typen der Funktionsparameter die nötigen Templateparameter ab, setzt diese in die Platzhalter des Templates ein und gneriert dann den vollständigen Code.
2. Ebenfalls beim erstmaligen Aufruf, aber hier werden die Templateparmeter vom Programierer angegeben.
3. Durch explizite Deklaration mit entsprechenden Parametern wird die nötige Code-Generierung auch ohne Aufruf der Funktion angestossen.

9. Klassenattribute „static“

Klassenattribute haben bekanntlich für alle Objekte einer Klasse nur einen Wert. Bei Klassenattributen von Klassentemplates gilt folgende Besonderheit:

Klassenattribute von Klassentemplates haben für jede Instantiierung einen Wert. Das mit static versehene Argument gilt also für alle Objekte, die in der aus der Templateklasse generierte Klasse enthalten sind.

Beispiel:

```
template <class Type>
class Vektor {
...
private:
     static int anzahlDerObjekte;
     ...
};

template <class Type> int Vektor<Type>::anzahlDerObjekte = 0;
               //initialisieren des statischen Attributs
Vektor<float> v;         //Instantiierung mit float
Vektor<int> v;           //Instantiierung mit int
```

Dieses Beispiel enthält also zwei statische Variablen, die unabhängig von einander sind.

10. Argumente von Klassentemplates

???

11. Klassentemplates mit Vererbung

Klassentemplates können auch zusammen mit Vererbung eingesetzt werden:

- Klassentemplates können Spezialisierungen von normalen Klassen sein,
- Klassentemplates können Spezialisierungen von anderen Klassentemplates sein
- normale Klassen können Spezialisierungen von Instantiierungen von Klassentemplates sein

➔ ev. Beispiele aus Dummies

12. Funktionstemplates

Ein ähnliches Problem wie bei den Klassen (welches Problem / wo?) tritt auch bei Funktionen auf: wenn wir Funktionen für verschiedene Datentypen ihrer Parameter überladen, so ist häufig der Code gleich. Der einzige Unterschied besteht im Datentyp der Parameter und evt. internen Variablen.

Sie sind so aufgebaut:

```
template<class Type>
void feldcopy( Type a[], Type b[], int n )
{ ... }
```

Im Unterschied zu Klassentemplates werden die Funktionstemplates vom Compiler automatisch bentzt. Der Compiler sucht bei einem Funktionsaufruf die pasende Funktion nach folgender Regel:

1. Falls eine normale Funktion vorhanden ist, die mit ihrem Datentypen exakt passt, wird diese verwendet
2. Falls ein Funktionstemplate vorhanden ist, das bei Ersatz des / der Typparameter durch reale Typen exakt passt, wird dies verwendet
3. Ansonsten wird nach den üblichen Regeln mit Typkonversionen nach einer normalen Funktion gesucht

Beispiel anhand einer Funktion `swap`, die die Werte ihrer beiden Argumente vertauscht:

```
template<class Type>
void swap( Type& x, Type& y )
{
```

```
    Type temp ;
    temp = x ;
    x = y ;
    y = temp ;
}
```

Die Funktion `swap` kann für alle eingebauten und selbstdefinierten Datentypen benutzt werden, falls bei diesen die gebrauchten Operatoren definiert sind. Die Funktion kann aber nicht für Felder benutzt werden. Falls dies doch erforderlich ist, muss die Funktion auch für Felder umgesetzt werden. Ein weiteres Template ist nicht möglich.
In diesem Beispiel wird die zusätzliche Funktion für ein Feld von `char` umgesetzt:

```
void swap( char s1[], char s2[] )
{
    int maxLen;
    if( strlen(s1) > strlen(s2) ) {
        maxLen = strlen(s1);
    } else {
        maxLen = strlen(s2);
    }
    char* temp = new char[maxLen + 1]; //+1 ist nötig wegen \0
    strcpy( temp, s1 );
    strcpy( s1, s2 );
    strcpy( s2, temp );
    delete[] temp;
}
```

13. Compilieren von Templates

Der Compiler stellt erst beim Linken fest, dass eine bestimmte Instantiierung eines Templates benötigt wird. Er muss diese dann herstellen und compilieren, bivor er mit dem Linken fortfahren kann.
Sorgfältiges Studium der Dokumentation zum Compiler ist daher erforderlich, wenn das Übersetzen von Programmen mit Templates nicht auf Anhieb gelingt.
Die vom Compiler herausgegebenen Fehler sind dabei vielmals nur indirekt ein Hinweis auf den Programmierfehler.