

Programmiertechnik (C++)

Prof. Dr. Stefan Enderle
NTA Isny

5. Templates

5.1 Funktions-Templates

- **Problem:**

In typgebundenen Programmiersprachen kommt es häufig vor, dass „gleiche“ Funktionen mehrfach implementiert werden, da sie für verschiedene Typen zur Verfügung stehen sollen.

- **Lösung:**

Mit Hilfe von *Templates* (*Schablonen*) können Funktionen definiert werden, die anstatt auf einem bestimmten Datentyp zu arbeiten, einen Platzhalter benutzen. Dieser Parameter wird erst später mit einem tatsächlichen Typ gefüllt.

Beispiel

- **Beispiel:**
Vertauschen zweier Integer:

```
void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Gleicher Code auch bei float, double, char, string, ...

Funktions-Template

- Schreibweise als Funktions-Template:

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

- Für T kann nun jeder Typ eingesetzt werden, vorausgesetzt, der Zuweisungsoperator `operator=` existiert.

Instanziierung

- Mit der Definition eines Funktions-Templates wird noch *keine* konkrete Funktion erzeugt!
- Der Maschinencode wird erst *dann* erzeugt, wenn die Funktion für einen bestimmten Typ zum ersten Mal benötigt wird.
- Wird die Funktion später *wieder* für diesen Typ benötigt, kann die bereits erstellte Funktion vom Compiler aufgerufen werden.

Instanziierung

- Beispiel:

```
int a=10;  
int b=5;  
swap(a,b);    // vertauscht a und b
```

```
string x="Hallo";  
string y="Welt";  
swap(x,y);    // vertauscht x und y
```

- Bemerkung: Das Funktions-Template swap ist im Namensbereich std bereits definiert!

Template + inline

- **Template + inline:**
Bei kurzen Template-Funktionen wie swap() kann zusätzlich „inline“ benutzt werden.
- Der „Funktionscode“ wird dann direkt in das Programm eingesetzt.

- **Beispiel:**

```
template <typename T>
inline void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Verschachteln

- **Verschachteln von Templates:**

Funktions-Templates dürfen geschachtelt werden.

D.h., ein Funktions-Template darf wieder ein Funktions-Template aufrufen.

- **Beispiel:**

```
template <typename T>
void sort(T arr[], int len)
{
    for (int i=0; i<len-1; i++) {
        int k = i;
        for (int j=i+1; j<len; j++)
            if (arr[j]<arr[k]) k=j;
        swap(arr[i], arr[k]);        // Aufruf swap
    }
}
```

Typprüfung

- **Typübereinstimmung:**
Der Compiler kann ein Template nur erzeugen, wenn die Typen **genau** übereinstimmen. Es wird nicht einmal z.B. automatisch von int nach long gecastet!
- **Beispiel:**

```
template <typename T>
T max(T a, T b) { return a>b? a : b; }

long x = 5;
long erg = max(x,10); // Fehler, da 10 int
```
- Hier wurde keine Funktion `max(long, int)` gefunden!

Explizite Generierung

- Ein Funktions-Template kann allerdings **explizit** generiert werden, damit die Funktion gefunden wird:

- **Beispiel:**

```
template <typename T>  
T max(T a, T b) { return a>b? a : b; }
```

```
long max(long, long); // Deklaration!!
```

```
long x = 5;
```

```
long erg = max(x, 10); // Nun OK
```

- Hier wurde die Funktion `max(long, long)` gefunden und 10 dann auf den Typ `long` gecastet.

Explizite Angabe der Template-Parameter

- Selbst wenn alle möglichen Template-Funktionen bereits erstellt wurden, ist manchmal der aktuell angegebene Typ *nicht eindeutig* auflösbar:

- **Beispiel:**

```
template <typename T1, typename T2>  
T1 f(T2 value) { ... }
```

```
f(1); // nicht klar
```

```
double d = f(2.5); // auch nicht
```

- **Explizite Angabe der Template-Parameter:**

```
f<double>(1); // klar: T1=double, T2=int
```

```
double d = f<int>(2.5); // T1=int, T2=double
```

Explizite Angabe der Template-Parameter

- Bei explizit angegebenen Template-Parametern reicht es aus, die uneindeutigen Typen anzugeben.
- Die restlichen Typen können automatisch aufgelöst werden.
- Allerdings müssen die expliziten Typen von links nach rechts angegeben sein!

Spezialisierung

- Aus folgenden Gründen kann ein Funktions-Template nicht für alle Typen einsetzbar sein:
 - Das Template enthält Anweisungen, die für den Datentyp nicht existieren
 - Die allgemeine Lösung liefert für den Datentyp kein sinnvolles Ergebnis
 - Für den Typ gibt es eine bessere Lösung.
- Beispiel:

```
max („Anna“, „Bernd“);  
    // Liefert größere Speicheradresse !!  
    // da Anna und Bernd const char* sind
```

Spezialisierung

- In diesen Fällen ist es möglich, eine *Spezialisierung* zu definieren.
- **Beispiel:**

```
// Allgemein //
template <typename T>
T max(T a, T b) { return a>b? a : b; }

// Spezialisierung //
const char* max(const char* a, const char* b)
{
    if (strcmp(a,b) > 0) return a;
    else return b;
}

maxStr = max(„Anna“, „Bernd“); // OK
```

Überladen

- Template-Funktionen können wie normale Funktionen auch „überladen“ werden. D.h., dem gleichen Funktionsnamen können verschiedene Parameterlisten zugeordnet sein.
- Beispiel:

```
template<typename T>  
void f(T x) { ... } // Ein Parameter
```

```
template<typename T>  
void f(T x, int) { ... } // Zwei Parameter
```

Überladen

- In folgendem Aufruf wird dann Template 1 verwendet:

`f(5);`

- hier Template 2:

`f(5, 2);`

5.2 Klassen-Templates

- **Problem:**
Häufig werden Container-Klassen benötigt, die für beliebige Typen funktionieren sollen.
Früher wurde dies oft mit Hilfe von `void*` gelöst, was aber zu unschönem Code und verlorener Typsicherheit führte.
- **Lösung:**
Mit Hilfe von *Klassen-Templates* können Klassen definiert werden, die einen unbestimmten Datentyp benutzen. Dieser unbestimmte Typ wird erst später mit einem tatsächlichen Typ gefüllt.

Beispiel

- Beispiel: Klasse „Stack“ für beliebige Typen T

```
template <class T>
class Stack {

private:
    T*  daten;           // Datenbereich des Stacks
    int index;         // Aktueller Index
    int size;          // Größe des Stacks

public:
    Stack(int s); // Konstruktor mit Größe
    void push(T item); // Element auf Stack "pushen"
    T top() const; // Element lesen
    T pop(); // Element lesen und löschen
    bool empty(); // Stack leer?
};
```

Instanziierung

- Wie bei den Funktions-Templates wird auch durch ein Klassen-Template noch keine konkrete Klasse erzeugt!
- Der Maschinencode wird erst dann erzeugt, wenn zum ersten mal eine Instanz der Template-Klasse angelegt wird.
- Beachte: Das Überladen von Klassen ist nicht (wie bei Funktionen) möglich.

Instanziierung

- Unterschied: Die Instanziierung muss explizit erfolgen!
- Beispiel:

```
Stack<int>      intStack;  
Stack<double>  doubleStack;
```

Methoden

- Methoden einer Template Klasse sind automatisch Template-Funktionen:

```
template <class T>
Stack<T>::Stack(int s) // Konstruktor
{
    index = -1;
    size(s);
    daten = new T[size]; // Stack als Array anlegen
}
```

```
template <class T>
void Stack<T>::push(T item) // Element auflegen
{
    if (index < (size-1)) {
        index++;
        daten[index] = item;
    }
}
```

Instanziierung (2)

- Achtung: Bei der Generierung einer Template-Klasse werden *nicht sofort* alle Methoden generiert. Es werden nur *die* Methoden generiert, die auch aufgerufen werden.
- Vorteil: Verwendet eine Methode Eigenschaften, die der Template-Typ nicht zur Verfügung stellt, so ist dies trotzdem korrekt, so lange diese Methode nicht verwendet wird.

Instanziierung (2)

```
template <class T>
class A
{
public:
    T value;

    A();
    bool isZero();
};
```

```
template <class T>
bool A<T>::isZero()
{
    if (value == 0) return true;
    else return false;
}
```

Instanziierung (2)

```
class B  
{  
};
```

```
A<int> intObjekt; // OK
```

```
A<B> bObjekt; // Auch OK, da isZero()  
// nicht benutzt wird
```

```
intObjekt.isZero(); // OK
```

```
bObjekt.isZero(); // Fehler !
```

Spezialisierung

- Wie auch bei Funktions-Templates, kann es vorkommen, dass aus folgenden Gründen ein Klassen-Template nicht für alle Typen einsetzbar ist:
 - Die Template-Klasse enthält Anweisungen, die für den Datentyp nicht existieren
 - Die allgemeine Lösung liefert für den Datentyp kein sinnvolles Ergebnis
 - Für den Typ gibt es eine bessere Lösung.

Explizite Spezialisierung

- In diesen Fällen ist es ebenfalls möglich, eine *Spezialisierung* zu definieren.
- Beispiel:

```
template <class T>
class A
{
    A(T val);
    ...
};
```

```
template <>
class A<double>
{
    A(const T&); // Darf sich unterscheiden
    ...           // T ist bekannt !
};
```

Partielle Spezialisierung

- Eine Spezialisierung für eine *Gruppe* von Typen ist auch möglich.
- Beispiel:

```
template <class T>
class A
{
    A(T val);
    ...
};
```

```
template <class T>
class A<T*>
{
    A(T val);
    ...
};
```

Spezialisierung

- Welche Realisierung wird gewählt?

```
A<int> intA; // Allgemeine Klasse
```

```
A<int*> intPointerA; // Partielle Spezialisierung
```

```
A<double> doubleA; // Explizite Spezialisierung
```