Programmieren - C++ Funktions-Templates

Reiner Nitsch
☏ 8417
✉ r.nitsch@fbi.h-da.de

Was sind Templates?

- **C++ Templates** ermöglichen **generische Programmierung**. Das ist Programmierung unabhängig vom speziellen Objekt-Typ
- Templates sind **Schablonen** mit **parametrisierten Datentypen** d.h. mit Platzhaltern für Typen (Typ-Parameter oder Proxy-Parameter): So wie das Laufzeitsystem beim Funktionsaufruf die Aktual-Parameter an Stelle der in der Deklaration benannten Formal-Parameter einsetzt, werden bei Templates **vom Compiler zur Compilezeit** die Aktual-Typen an Stelle der deklarierten Formal-Typparameter eingesetzt.
- C++ unterscheidet zwischen **Funktionstemplates** (dieses Kapitel) und **Klassentemplates** (später)
- Beispiel für Funktionstemplates: Vertauschen von `int`-, `double`-, `string`- oder `Konto`-Objekten

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

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

Anforderungsprofil: Welche Ansprüche stellt der Algorithmus an die Typen 'int' und 'Konto'?

- Soll die Operation "swap" mit dem Konzept der Fkt-Überladung für m verschiedene Datentypen realisiert werden, so muß die Funktion **m-fach** überladen werden (ggf. mittels Cut&Paste und Typanpassung; Fehleranfällig).
- Mit dem C++-Template-Konzept muß die Funktion nur **einmal** definiert werden, um für beliebige Typen, die dem **Anforderungsprofil** genügen, anwendbar zu sein.

Beispiel 1: Funktion vertausche als Funktionstemplate

Beispiel 1:

Deklariert Funktion als Template, das vom Compiler erst bei Bedarf übersetzt wird.

Proxy-Typ T kann auch einfacher Datentyp sein! Auch mehrere kommaseparierte Typ-parameter sind möglich

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

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

Achtung!
Copy-Konstruktor und Zuweisung müssen für Typ T definiert sein!

Compiler ersetzt jedes `T` durch `int`!

Referenzen oder Pointer werden hier nicht unterstützt

Anwendung des Templates swap

```
void main()
{
  int a=5, b=6;
  swap( a , b );

  double x=2.0, y=3.0;
  swap( x , y );
  swap( a , x );
}
```

Compiler erzeugt erst hier Code für Funktion swap mit Aktual-Parametertyp `int` und deren Aufruf.

Hier wird Funktion swap mit Aktual-Parametertyp `double` gemäß Schablone erneut compiliert.

Fehler: Compiler verlangt hier gleiche Typen, weil auch die Proxy-Typen gleich sind (Kein autom. impliziter Type-Cast bei Templates).

Überladene Funktionstemplates

- Für Argumente von Template-Funktionen gibt es keine implizite Typumwandlung
- Explizite Typumwandlung wird unterstützt

```
template <typename T> sqrt(T);

void f( int i, double d, complex z) {
  complex z1 = sqrt(i); // sqrt(int) per Deduktion
  complex z2 = sqrt(d); // sqrt(double) per Deduktion
  complex z3 = sqrt<>(i); // sqrt(int) per Deduktion
  complex z4 = sqrt(z); // sqrt(complex) per Deduktion

  complex z5 = sqrt( double(i) ); // sqrt(double) per Deduktion
  complex z6 = sqrt<double>(i); // sqrt<double> ohne Deduktion
}
```

Hier wird immer die zum Argumenttyp passende sqrt-Funktion generiert

Wird etwas anderes gewünscht,
muss explizit gecastet oder deduziert werden.

Überladene Funktionstemplates

- Überladungsmöglichkeiten für Template-Funktionen:
  - durch Funktion gleichen Namens
  - durch andere Template-Funktion gleichen Namens
- Aufruf-Auflösung bei Überladung in 3 Schritten:
  1. Suche nach Funktion mit exakt gleicher Signatur
  2. Suche nach Template-Funktion mit exakt passender Argumentliste
  3. Suche nach einer überladenen Funktion, die nach einem impliziten Cast eines oder mehrere Argumente aufgerufen werden kann.

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

void f( int a, int b, char c1, char c2 ) {
  int m1 = max(a,b);          // max(int,int)
  char m2 = max(c1,c2);       // max(char,char)
  int m3 = max(a,c1);         // Fehler: max(int,char) kann nicht generiert werden
}                             // da keine Casts versucht werden (Schritt 2)
```

- An dieser Fehlerstelle wird erfolglos Schritt 3 versucht. Abhilfe:

```
template < typename T > T max(T a, T b) { return a<b ? b : a; }
int max(int a, int b) { return a<b ? b : a; }
void f( int a, int b, char c1, char c2 ) {
  // …
  int m3 = max(a,c1);         // Jetzt OK! max(int,int) wird aufgerufen (Schritt 3)
}
```

Beispiel 2: Funktion `selectionSort` als Funktionstemplate

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
//Sortieren im Bereich [links,rechts)

void selectionSort( int a[], int links, int rechts) {
   int i, j, min;
   for( i = links; i < rechts ; i++) {
      min = i;
      for( j=i+1; j < rechts; j++)
         if( a[j] < a[min] )  min = j;
      swap( a[i], a[min] );
   }
}
```

```
//Sortieren im Bereich [links,rechts)
template < typename T >
void selectionSort( T a, int links, int rechts) {
   int i, j, min;
   for( i = links; i < rechts ; i++) {
      min = i;
      for( j=i+1; j < rechts; j++)
         if( a[j] < a[min] )  min = j;
      swap( a[i], a[min]);
   }
}
```

An Stelle des Schlüsselworts '`typename`' kann auch das '`class`' verwendet werden.

oder **besser**:

```
//Sortieren im Bereich [links,rechts)
template < typename T >
void selectionSort( T a, int links, int rechts ){
   int i, j, min;
   for(  i = links; i < rechts ; i++ ) {
      min = i;
      for(  j=i+1; j < rechts; j++ )
         if( compare( a[j], a[min]) ) min = j;
      swap( a[i], a[min] );
   }
}
```

Begründung:

- `operator<` sorgt i.A: für eine aufsteigende Sortierung. Soll z.B. nach den Beträgen von `int`-Werten sortiert werden, so ist dies mit `operator<` nicht möglich, ohne den Sortieralgorithmus zu ändern. Deshalb delegiert man den Entscheid über die korrekte Reihung besser an eine allgemeine Vergleichsfunktion **'compare'**, die Bestandteil des Elementtyps ist. Damit sind Elementtyp und Sortier-Algorithmus weitgehend entkoppelt (**generische Programmierung**) .

```
bool compare( const int& a, const int& b ) { return a<b; }
bool compare( const int& a, const int& b ) { return abs(a)<abs(b); }
```

Aufgabe 1

- Implementieren Sie die Funktionen `less`, `abs` und `lessAbs` als Template.

```
bool less( int a, int b ) { return a<b; }
int abs( int a ) { return a>0 ? a : -a; }
bool lessAbs( const int& a, const int& b ) { return abs(a)<abs(b); }
```

Aufgabe 2

- Implementieren Sie eine Such-Funktion als Template mit folgenden Eigenschaften:
  - Aufruf: `unsigned int fundStelle = find(c, first, last, value );`
  - Rückgabe des Index `i` des ersten Elements im Containers `c` im Bereich `[first,last)` für das die folgende Bedingung eingehalten wird: `c[i] == value`
  - Rückgabe `last` wenn `value` nicht enthalten ist.
  - Komplexität: Höchstens `last - first` Anwendungen der Predicate-Funktion.
- Schreiben Sie eine Test-Anwendung, die `find` mit einem `int`-C-Array und einem `vector<string>`-Container anwendet.

Aufgabe 2 - Lösung

Aufgabe 3 (Hausaufgabe)

- Implementieren Sie die Funktion `max` als Template und schreiben Sie eine Test-Anwendung, die `int`- und `Konto`-Objekte (Reihung basierend auf `Konto::stand`) verwendet .

```
Typ& max( Typ , Typ )
```

So, das war´s erst mal!