LANDESBERUFSSCHULE 4 SALZBURG

Informatik

Template – generisches Programmieren

LBS 4

Inhalt

Der Weg zum eigenen Template

Templates werden als generische Programmierung bezeichnet, weil der Datentyp nicht entscheidend für die Funktion der Vorlage ist. Diese werden zur Compilezeit abgearbeitet. Der Compiler ersetzt zu diesem Zeitpunkt die parametrisierten Werte mit den definierten Datentypen.

Ein Template wird mit dem Schlüsselwort ***template*** eingeleitet. Dahinter folgt in spitzen Klammern der Datentyp als Platzhalter. Das Schlüsselwort ***typename*** oder ***class*** definiert den Platzhalter für einen beliebigen Datentyp.

Funktionstemplate

Als Beispiel wird hier die min(a,b) Funktion verwendet. Diese gibt den niedrigsten Wert zurück.

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

Ein Template kann auch mit Referenzen oder anderen Modifikationen arbeiten.

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

Die Verwendung zur Laufzeit geschieht genauso, als ob man Templates aus der STL verwendet. Es wird der Name des Template mit dem Datentyp angegeben.

```
std::cout << my_min<int>(23,56) << std::endl;
```

Klassentemplate

Sehr häufig wird ein Klassentemplate verwendet. Die Klassen sind von ihren Attributtypen und Methodenparameter abhängig. Als Beispiel wird hier die allgemeine Form eines Arrays verwendet.

```
template <typename T, unsigned n>
class MyArray
{
private:
   T data[n];
public:
   enum { size = n };
   T& operator[]( int i ) { return data[i]; }};
```

Das Array kann mit einem beliebigen Datentyp initialisiert werden der einen Konstruktor besitzt und kopiert werden kann. Der Datentyp ***size*** ist eine Konstante die keinen Speicherplatz belegt. Bei der Instanziierung müssen die Parameter ebenfalls in spitzen Klammern angegeben werden.

```
MyArray<int, 7> test_array;//statisch
MyArray<int, 7> *pa = new MyArray<int, 7>();//dynmiach

for(int i = 0; i < test_array.size; i++){
   test_array[i]= i*i+3;
   (*pa)[i] = i*i+4;
}
for(int i = 0; i < test_array.size; i++){
   std::cout << test_array[i] << std::endl;
   std::cout << (*pa)[i] << std::endl;
}
```

Klassen haben aber nicht nur Inline-Methoden sondern auch welche die außerhalb der Klasse definiert werden. Für diese werden die Templatedefinitionen unabhängig von der Klasse definiert.

Methodentemplates

Das Array kann noch mit Methoden erweitert werden. In diesem Beispiel wird die min() und max() Methode implementiert. Diese geben den höchsten oder niedrigsten Wert zurück. Der Prototyp wird um die Methoden ergänzt.

```
template <typename T, unsigned n>
class MyArray
{
private:
   T data[n];
public:
   enum { size = n };
   T& operator[]( int i );
   const T& max() const;
   const T& min() const;
};
```

Im Klassentemplate befinden sich die Prototypen, die Implementierung erfolgt außerhalb des Template.

```
template <typename T, unsigned n>
const T& MyArray<T,n>::max() const
{
   const T* p = &data[0];
   for( int i = 1; i < size; ++i )
      if( *p < data[i] )
         p = &data[i];
   return *p;
}
```

Beispiel: eigene Funktion zum Template:

Im nachstehenden Beispiel wird ein Funktionstemplate find() welches auch in der C ++ Standardbibliothek vorhanden ist nachprogrammiert und Schritt für Schritt zu einem Template abstrahiert.

Prototypen

Es wird ein Container vom Typ Array verwendet. Zusätzlich wird ein Typname *Lbs4Iterator* mit *using* eingeführt. Der Iterator entspricht hier einem Zeiger.

Die unterschiedliche Verwendung ist nachstehend dargestellt:

```
typedef std::vector<std::string>::iterator StringVectorIterator;
using StringVectorIterator = std::vector<std::string>::iterator;
```

Seit C++11 kann man statt typedef das Schlüsselwort using verwenden

Der Prototype find() besitzt zwei Parameter vom Typ: Lbs4Iterator und einen Integer als gesuchten Wert. Die Rückgabe ist ebenfalls ein Zeiger vom type Lbs4Iterator

```
include <iostream>
using namespace std;
// neuer Typname Lbs4Iterator für `Zeiger auf const int'
using Lbs4Iterator = const int*;
// Prototyp des Algorithmus
Lbs4Iterator find(Lbs4Iterator begin, Lbs4Iterator end, int wert);
```

Implementation

Die Implementation ist sehr einfach gehalten, erfüllt aber den Zweck

```
// Implementation
Lbs4Iterator find(Lbs4Iterator begin, Lbs4Iterator end, int wert) {
  while (begin != end && *begin != wert) { // Dereferenzierung und Objektvergleich
    ++begin;                 // nächste Stelle
  }
  return begin;
}
```

Main-Methode

In der main()-Methode wird das Array, die Anzahl der möglichen Elemente, sowie der Zeiger auf das erste Element und ein Zeiger nach dem letzten Element erstellt.

```
int main() {
  constexpr int ANZAHL{100};
  int class_room[ANZAHL];// Container definieren
  Lbs4Iterator begin{class_room}; // Zeiger auf den Anfang
  //  Position NACH dem letzten Element
  Lbs4Iterator end{class_room + ANZAHL};

  // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
  for (size_t i = 0; i < ANZAHL; ++i) {
    class_room[i] = 2 * i;
  }
  int zahl{0};
  while (zahl != -1) {
    cout << "Katalognummer eingeben (-1 = Ende):";
    cin >> zahl;
    if (zahl != -1) { // weitermachen?
      Lbs4Iterator position = find(begin, end, zahl);
      if (position != end) {
        cout << "gefunden an Position " << (position - begin) << endl;
      } else {
        cout << zahl << " nicht gefunden!" << endl;
      }
    }
  }
}
```

Eigenes Template ohne STL

Als nächstes wird statt der Implementation der Funktion ein Template erstellt. Wichtig sind im Prototyp die Parameter für den Iterator und der Typname des Datentyps.
T steht für den verwendeten Datentyp und Iterator für den verwendeten Zeiger.

```
// Prototyp des Algorithmus ersetzt durch Template
template <class Iterator, typename T>
Iterator find(Iterator begin, Iterator end, const T &wert) {
  while (begin != end                  // Zeigervergleich
        && *begin != wert) {           // Dereferenzierung und Objektvergleich
    ++begin;                           // nächste Stelle
  }
  return begin;
}
```

Template aus der STL „vector"

Im dritten Beispiel wird ein Template aus der STL verwendet. Hier wird in der ersten Zeile der Container vector mit dem Datentyp int als class_room initialisiert.

```
#include <iostream>
#include <vector> //container
#include <algorithm> //
using namespace std;
int main() {
  vector<int> class_room(100); // Container definieren
  // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
  for (size_t i = 0; i < class_room.size(); ++i) {
    class_room[i] = 2 * i;
  }
  int zahl{0};
  while (zahl != -1) {
    cout << " gesuchte Zahl eingeben (-1 = Ende):";
    cin >> zahl;
    if (zahl != -1) { // weitermachen?
auto position = find(class_room.begin(), class_room.end(), zahl);
if (position != class_room.end()) {
       cout << "gefunden an Position " << (position - class_room.begin()) << endl;
     }
else {
       cout << zahl << " nicht gefunden!" << endl;
     }
   }
  }
}
```

Aufgabe:

Wählen Sie eine der Aufgaben aus:

- Verwenden Sie die Headerdatei Ort vom Moodle. Erstellen Sie ein Template der Klasse mit den Methoden. Testen Sie ihr Template in der main-Funktion. (max. 75%)
- Erstellen Sie ein Template von ihrer Klasse VekQueue. Es soll nicht nur die Klasse, sondern auch die Methoden generisch gemacht werden. Testen Sie ihr Template in der main-Funktion.

Geben Sie das Template mit den Methoden am Moodle ab!