Templates

Thomas Hausberger, Matthias Panny
Ursprünglich erstellt von Sebastian Stabinger

SS2022

Problem

Quadrieren von Zahlen in C

Beispiel

```c
short quad_short(short a) { return a * a; }
int quad_int(int a) { return a * a; }
float quad_float(float a) { return a * a; }
double quad_double(double a) { return a * a; }

#include <stdio.h>

int main() {
  printf("Int: %d\n", quad_int(23));
  printf("Float: %f\n", quad_float(24.12));
}
```

- Für jeden Datentyp brauchen wir eine eigene Funktion
- Die Funktionen müssen alle unterschiedlich benannt sein
- Aber eigentlich machen wir in jeder Funktion das selbe!

Quadrieren von Zahlen in C++

- Bis jetzt haben wir Funktionsüberladung kennen gelernt. Das macht die Situation zumindest in der Verwendung der Funktionen besser: Wir können alle Funktionen gleich benennen

Beispiel

```
#include <iostream>
using namespace std;

short quad(short a) { return a * a; }
int quad(int a) { return a * a; }
float quad(float a) { return a * a; }
double quad(double a) { return a * a; }

int main() {
  cout << "Int: " << quad(23) << endl;
  cout << "Float: " << quad(24.12) << endl;
}
```

- Wir machen immer noch das selbe in allen Funktionen!

Lösung: Templates

Grundlagen

- Wir können mittels Templates einen Typ als Parameter übergeben
- Möglich bei Funktionen oder Klassen
- Geschieht mittels `template <typename T>` vor der Funktion oder Klasse
- T ist dann ein Platzhalter für einen beliebigen Typ den wir übergeben können
- Templateparameter werden in spitzen Klammern statt runder Klammern übergeben. z.B. `quad<int>(i)` um einer Templatefunktion `quad` als Typ `int` und als normalen Parameter `i` zu übergeben.

Hinweis

Aus historischen Gründen kann man statt `template <typename T>` auch `template <class T>` schreiben.

Funktionstemplates

Funktionstemplates werden verwendet um bei einer Funktion einen Typ als Parameter übergeben zu können

Beispiel für quad

```
#include <iostream>
using namespace std;

template <typename T> T quad(T a) { return a * a; }

int main() {
  cout << "Int: " << quad<int>(23) << endl;
  cout << "Float: " << quad<float>(24.12) << endl;
}
```

- Aufruf: `f< Templateparameter >( Normale Parameter )`

Was passiert hinter den Kulissen?

- Wir haben quad folgendermaßen definiert:

```
template <typename T> T quad(T a) { return a * a; }
```

- Irgendwo im Code rufen wir quad<int>(42); auf. Der Compiler erzeugt automatisch eine Funktion mit konkretem Typ mit Hilfe unseres Funktionstemplates:

```
template <typename T> T quad(T a) { return a * a; }
// T wird durch quad<int>(42) der Typ int zugewiesen --->
int quad(int a) { return a * a; }
// Diese Funktion wird dann ganz normal auf den Wert 42 angewendet
```

- Templatefunktionen sind daher bei der Ausführung nicht langsamer als "ganz normale" Funktionen
- Beim Compilieren wird also tatsächlich neuer Code erzeugt

Hinweis

Template kann man als Schablone ins Deutsche übersetzten: Wir haben eine Schablone für Funktionen und Klassen mit deren Hilfe wir spezifische Versionen von Funktionen und Klassen erzeugen können.

Typinferenz

- Wir haben `quad` folgendermaßen definiert:

```
template <typename T> T quad(T a) { return a * a; }
```

- Es ist offensichtlich etwas unpraktisch den gewünschten Typ bei einer Templatefunktion immer explizit in spitzen Klammern angeben zu müssen
- Wenn wir `quad<int>(42)` aufrufen ist dem Compiler ja bereits durch den Parameter `42` bekannt, dass `T` ein Integer sein muss
- Tatsächlich müssen wir den Typ nicht explizit angeben, falls dem Compiler der Typ bekannt sein kann (was bei Funktionstemplates oft der Fall ist)
- Wir können also einfach `quad(42)` schreiben

Beispiel max

```cpp
#include <iostream>
using namespace std;

template <typename T> T t_max(T a, T b) { // t_max weil es max schon gibt
  T ergebnis; // Wir koennen auch Variablen vom unbekannten Typ erzeugen
  if (a > b)
    ergebnis = a;
  else
    ergebnis = b;

  return ergebnis;
}

int main() {
  int a = 23, b = 42;
  double c = 34.2, d = 1.3;
  cout << "max int = " << t_max(a, b) << endl;
  cout << "max double = " << t_max(c, d) << endl;
  // Was passiert wenn wir Typen mixen?
  // cout << "max int/double = " << t_max(a, c) << endl;
}
```

Beispiel max — Gemischte Typen

Was passiert beim Aufruf von Folgendem?

```
cout << "max int/double = " << t_max(a, c) << endl;
```

Ausgabe Compiler

```
max.cpp: In function ‘int main()’:
max.cpp:20:44: error: no matching function for call to
               ‘t_max(int&, double&)’
   cout << "max int/double = " << t_max(a, c) << endl;
                                           ^
max.cpp:20:44: note: candidate is:
max.cpp:4:25: note: template<class T> T t_max(T, T)
 template <typename T> T t_max(T a, T b) {
                         ^
max.cpp:4:25: note:   template argument deduction/substitution failed:
max.cpp:20:44: note:   deduced conflicting types for parameter ‘T’
                       (‘int’ and ‘double’)
   cout << "max int/double = " << t_max(a, c) << endl;
                                           ^
```

Klassentemplates — Beispiel

- Funktionieren genauso wie Funktionstemplates. Man kann Typen als Parameter für eine Instanz einer Klasse übergeben

```
#include <iostream>
using namespace std;

template <typename Sum, typename Count> class average {
private:
  Sum _sum = 0;
  Count _count = 0;

public:
  void add(Sum val) { _sum += val; _count++; }

  Sum avg() { if (_count == 0) return 0; else return _sum / _count; }
};

int main() {
  average<float, int> a;
  cout << "a " << a.avg() << endl;
  a.add(23);
  a.add(27);
  cout << "a " << a.avg() << endl;
}
```

Abschluss

- Wie bereits gesagt kann man sich Templates als Schablonen vorstellen. In diesen Schablonen werden Platzhalter für Typen durch einen übergebenen Typer ersetzt
- Beim Compilieren wird tatsächlich neuer Code generiert. Das führt dazu, dass die Compilezeiten länger werden.
- Wir haben Templates z.B. schon bei `vector` gesehen. Der zu speichernde Typ wird in spitzen Klammern angegeben:

```
vector<int> v; v.push_back(23);
```

Hinweis

Templates sind bei weitem komplexer und mächtiger als es hier den Anschein hat. Es handelt sich im Prinzip um eine eigene Programmiersprache (das war eigentlich nicht so beabsichtigt). Wer sich den Wahnsinn des "Template Metaprogrammings" anschauen will kann z.B. auf der Wikipedia Seite anfangen.