Ein kleiner Blick auf die generische Programmierung

TgZero
Technik.Blosbasis.net

June 3, 2013

Inhaltsverzeichnis

1 Vorwort **3**

2 Ein kleines Beispiel **3**

3 Templates **3**

4 Verschiedene Datentypen **4**

5 Variadic Templates **5**

6 Unterschied zwischen den Programmiersprachen **6**

7 Abschließende Worte **7**

1 Vorwort

Generische Programmierung sollte für den heutigen modernen Entwickler durchaus zur Trickkiste gehören. Es geht darum, Funktionen, Strukturen oder Klassen möglichst allgemein zu entwerfen, ohne genauere Datentypen zu verwenden und diese praktisch zur Laufzeit/ beim Kompilieren zu generieren.

2 Ein kleines Beispiel

Nehmen wir an, wir würden uns einen Taschenrechner schreiben der zwei Werte miteinander addieren soll.Dabei befinden wir uns in einer getypten Programmiersprache (C++ / Java).
Wenn wir nun Ganzzahlen betrachten, sollte dies kein Problem sein - ein einfacher Integer sollte durchaus für unsere Zwecke reichen. Doch wie handhaben wir es, wenn der Benutzer eine Zahl mit beliebig vielen Nachkommastellen angibt? Benutzten wir hier eine Typumwandlung und ignorieren die Nachkommastellen, oder schreiben wir eine zweite Funktion AddKomma?
Und genau hier kommt unsere generische Programmierung ins Spiel...

3 Templates

Templates werden in Programmiersprachen als "Platzhalter" benutzt, wobei erst zur Übersetzungszeit/Laufzeit bekannt ist, welcher Datentyp für diesen Platzhalter eingesetzt wird.
Anstatt nun direkt den Datentyp Integer für die Funktion add zu nehmen, benutzen wir ein Template

Listing 1: template.cpp

```
1  template <typename T>
2
3  T Add(T val1, T val2)
4
5  {
6
7      return val1 + val2;
8
9  }
```

Nun können wir per

```
1  Add<float>(12,15.333)
```

zwei Variablen des gleichen Typs miteinander addieren, wobei wir zwischen den <>einen Datentyp angeben können.Um verschiedene Datentypen miteinander zu addieren und dabei automatisch die Nachkommastellen zu berücksichtigen ist ein kleiner Trick notwendig. Wir benutzen std::ceil um zu überprüfen, ob eine Zahl eine Nachkommastelle besitzt. Anschließend rufen wir unsere Funktion "Add" mit dem Datentyp float auf. Dabei werden unsere beiden Datentypen zu float umgewandelt.

Listing 2: templatefull.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  template <typename T>
7  T Add(T val1, T val2);
8
9
10 int main()
11 {
12     float kommazahl = 10.99;
13     if(kommazahl < std::ceil(kommazahl))
14         std::cout << Add<float>(kommazahl,kommazahl) << std::endl;
15     else
16         std::cout << Add<int>(kommazahl,kommazahl) << std::endl;
17    return 0;
18 }
19
20 template <typename T>
21 T Add(T val1, T val2)
22 {
23     return val1 + val2;
24 }
```

4 Verschiedene Datentypen

Bisher hatten wir unsere Funktion "Add" so definiert, dass wir ihr nur zwei Parameter des gleichen Typs angeben konnten. Dies soll sich jetzt ändern und es ist denkbar einfach.
Anstatt

```
1  template<typename T>
2  T Add(T val1, T val2);
```

fügen wir einen weiteren typename hinzu:

```
1  template<typename T,typename F>
2  T Add(T val1, F val2);
```

Nun können wir zwei Variablen verschiedener Datentyps miteinander addieren, leider verlieren wir dennoch die Nachkommastellen, da bei dem oper-

ator+() automatisch eine Typumwandlung (zum Beispiel zu einem Integer) stattfindet.

5 Variadic Templates

Zum einen bieten Variadic Templates den Vorteil, dass diese Funktionen beliebig viele Parameter empfangen können und zum anderen diese Parameter auch einen anderen Typ haben dürfen. Ich übernehme mal das Beispiel von Rainer Grimm (C++11 - Der Leitfaden für Programmierer zum neuen Standard, 2012 Addison-Wesley Verlag) und zeige euch den gekürzten Code

Listing 3: variadic.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  #include <typeinfo>
4
5  template<typename T>
6  void PrintType(T Value)
7  {
8      std::cout << typeid(Value).name() << std::endl;
9  }
10
11
12 template<typename T>
13 void PrintInfo(T value)
14 {
15     PrintType(value);
16 }
17
18
19 template<typename First,typename ... Rest>
20 void PrintInfo(First first,Rest ...rest)
21 {
22     PrintType(first);
23     PrintInfo(rest...);
24 }
25
26 int main(int argc, char** argv)
27 {
28     PrintInfo(1,2,3,"true",true,13.66,(float) 13.99);
29     return 0;
30 }
```

Wie ihr durchaus richtig vermutet sind die drei Punkte das wesentliche bei den Variadic Templates.
Falls diese vor der Variable stehen

```
1  Rest ...rest
```

so wird der Rest der Parameter in die Variable rest gepackt. Falls diese wie bei dem rekursivem Funktionsaufruf wie bei PrintInfo stehen, so wird die Parameterliste entpackt.
Damit auch das letzte Element der Liste ausgelesen wird, muss die Funktion

mit nur einem Parameter definiert werden.
Um zurück auf unser Problem mit den Nachkommastellen zu kommen:
Man könnte nun eine rekursive Funktion mithilfe von Variadic Templates schreiben, die alle Parameter der Liste durchgeht und den "größten/gewünschten" Wert zurückgibt, schreiben. In unserem Beispiel wäre das eine Funktion, die zwischen int, double und float unterscheidet. Je nachdem welchen Rückgabewert diese Funktion besitzt, kann man alle Variablen die man addieren möchte in den gewünschten Datentyp umwandeln. Hier ein kleines Beispiel wie man den "größten" Datentyp von Zahlen ermitteln kann:

Listing 4: getbiggest.cpp

```
1  template<typename T>
2  char GetBiggest(char c,T value)
3  {
4          char n = *typeid(value).name();
5          // int < float < double
6          if(c == 'i' && (n == 'f' || n == 'd'))
7              c = n;
8
9          if(c == 'f' && n == 'd')
10             c = n;
11
12         return c;
13 }
14
15 template<typename First,typename ... Rest>
16 char GetBiggest(char c,First first,Rest ...rest)
17 {
18         c = GetBiggest(c,first);
19         c = GetBiggest(c,rest...);
20         return c;
21 }
```

Anschließend muss man nur per switch-Anweisung die passende Add-Funktion aufrufen.

6 Unterschied zwischen den Programmiersprachen

Bisher habe ich alle Beispiele in C++ (C++11) geschrieben, allerdings sind Templates nicht nur auf C++ beschränkt. Andere Programmiersprachen (wie Java) bieten dies ebenfalls an. Dennoch gibt es einen kleinen, aber feinen Unterschied. Während bei C++ die Generierung auf Kosten der Übersetzungszeit läuft, muss bei Java die Laufzeit her halten. Wenn wir in C++ eine Template-Funktion erstellen, überprüft der Compiler welche möglichen Datentypen bei der Funktion auftreten können, und generiert diese für uns. Damit sparen wir zwar Tipparbeit, allerdings der Compiler hat jede Menge zu tun. Bei Java hingegen wird nicht jede Funktion erneut generiert, sondern die JVM kümmert sich um die passenden Aufrufe zur Laufzeit.

Dadurch wird das Programm unter Java wesentlich langsamer, lässt sich allerdings schneller in Bytecode übersetzen.

7 Abschließende Worte

Ich muss zugeben, dass mir das am Anfang gewählte Beispiel zum Schluss einige Schwierigkeiten eingebrockt hat. Eigentlich sollte der Programmierer wissen, welche Datentypen er zu welcher Zeit benutzen muss und welche er getrost ignorieren kann. Da wir bei einer Funktion fast immer wissen, welche Datentypen wir als Parameter übergeben, ist es unnötig diese noch extra zu überprüfen (wie in dem Taschenrechnerbeispiel).
Dennoch hoffe ich, dass ihr Spaß beim Lesen hattet und etwas neues lernen konntet.

mfg
euer TgZero