C++ Templates - eine kleine Einführung

Fabian Scheler, Peter Ulbrich, Niko Böhm

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

20. Oktober 2008

Allgemein

- Templates ermöglichen generische Programmierung in C++

Generische Programmierung

- möglichst allgemeine Algorithmen und Datenstrukturen
- verzichte auf die Festlegung von Typen
- Anforderungen an Typen sind erlaubt
- *structural vs. nominative type system*

- zwei Ausprägungen
  - Funktionstemplates
  - Klassentemplates

Funktionstemplates

- generische Beschreibung von Algorithmen
- Deklaration/Definition
- Verwendung

Definition/Deklaration

- Deklaration

```
template <class class_name> void myfunc();
template <type type_name> void myfunc();
```

- Definition

```
template <class class_name> void myfunc() {
   // ...
}
```

- es sind auch mehrere Parameter erlaubt

```
template <class A, class B> void myfunc();
```

- gültige Parameterarten: Typen (auch Zeiger) und Ganzzahlen

Verwendung

- Beispiel: generische max()-Funktion

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

- Verwendung

```
int a = 5,b = 7,c;
c = max< int >(a,b);
```

- manchmal kann der Übersetzer den Typ *erraten*

```
int a = 5,b = 7,c;
c = max(a,b);
```

Klassentemplates

- generische Beschreibung von
  - Datenstrukturen und
  - zugehöriger Algorithmen
- Deklaration/Definition
- Definition von Methoden
- Verwendung

Deklaration/Definition

- Deklaration

```
template < type T, int I > class Tuple;
```

- Definition

```
template < type T, int I > class Tuple {
public:
  void put(T item, int pos);
  T    get(int pos);
  void sort();
};
```

- Parameterlisten: siehe Funktionstemplates

Definition von Methoden

- im Klassenrumpf

```
template < type T, int I > class Tuple {
public:
  void put(T item, int pos) { /* ... */ }
  // ...
};
```

- außerhalb des Klassenrumpfs

```
template < type T, int I >
void Tuple< T,I >::put(T item,int pos) {
  // ...
}
```

- Achtung: Trennung in Header (`.h`) und Implementierungseinheit (`.cc`) existiert für Templates in gängigen Toolchains noch nicht

Verwendung

- Beispiel

```
int a = 1,b = 2,c;
Tuple< int,5 > my_tuple;

my_tuple.put(a,0);
my_tuple.put(b,1);
my_tuple.put(4,2);

c = my_tuple.get(2);
```

- Templates ermöglichen typsichere Programmierung

```
float d = 0;
my_tuple.put(d,3); // compile time error
```

Template-Instantiierung

- Templates beschreiben noch keinen ausführbaren Code
- Übersetzer muss erst noch Lücken (Parameter) füllen
- diesen Vorgang bezeichnet man als *Instantiierung*
- Beispiel: generische max()-Funktion

Template

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

Instantiierung (T == int)

```
int max(int a,int b) {
  return a > b ? a : b;
}
```

- Wann findet die Instantiierung statt? - Bei der Verwendung

```
c = max< int >(a,b);
```

Template-Spezialisierung

- manchmal möchte man nicht alle Template-Parameter gleich behandeln
- Behandlung gewisser Sonderfälle
- Beispiel: Sortieren
  - Tupel der Länge 1 sind trivialerweise sortiert
- Lösung: man *spezialisiert* das Template

```
template< type T, int L > struct Sorter {
  void sort(T* array) { ... }
};

template< type T > struct Sorter< T,1 > {
  void sort(T* array) {}
}
```

vollständige vs. partielle Spezialisierung

- vollständige Spezialisierung legt alle Parameter fest

```
template<> struct Sorter< int,1 > {
  void sort(int* array) {}
}
```

- partielle Spezialisierung schränkt den Parameterraum ein

```
template< type T > struct Sorter< T,1 > {
  void sort(T* array) {}
}
```

- Funktionstemplates sind nicht partiell spezialisierbar!

Template Metaprogramming

- . . . C++ Templates sind noch viel mehr als das . . .
- C++ Templates sind eine funktionale Programmiersprache
  - eingebettet in C++
  - wird vom Übersetzer *ausgeführt*
  - geeignet zum schreiben von Metaprogrammen
  - geeignet für generative Programmierung

Metaprogramme

Ein *Metaprogramm* verarbeitet, modifiziert oder erzeugt ein oder mehrere andere Programme.

Generative Programmierung

Generative Programmierung ist ein Paradigma, dessen charakteristische Eigenschaft es ist, dass Quelltext durch einen Generator erzeugt wird.

Fakultät als Template Metaprogramm

```
volatile int c;

template< int I > struct Faculty {
  enum { result = I * Faculty< I - 1 >::result };
};

template<> struct Faculty< 1 > {
  enum { result = 1 };
};

int main(void) {
  c = Faculty< 5 >::result;
  return 0;
}
```

Fakultät als Template Metaprogramm

- die Fakultätsfunktion wird komplett im Übersetzer berechnet

Assembler-Dump der Funktion `main()`

```
<main>:
  lea     0x4(%esp),%ecx
  and     $0xfffffff0 ,%esp
  push    0xfffffffc(%ecx)
  push    %ebp
  mov     %esp,%ebp
  push    %ecx
  mov     $0x78 ,0x8049640
  mov     $0x0,%eax
  pop     %ecx
  pop     %ebp
  lea     0xfffffffc(%ecx),%esp
  ret
```

Hier gibt es noch viel mehr ...


Andrei Alexandrescu.
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley, 2001.


Krysztof Czarnecki and Ulrich W. Eisenecker.
Generative Programming. Methods, Tools and Applications.
Addison-Wesley, May 2000.


Stefan Kuhlins and Martin Schader.
Die C++-Standardbibliothek.
Springer-Verlag, 1999.


David Vandevoorde and Nicolai M. Josuttis.
C++ Templates: The Complete Guide.
Addison-Wesley, 2003.