



C++ Templates

Modern Techniques And Methods

Marcus "fritschy" Fritsch

8. Mai 2008

Agenda

- ▶ Grundlagen
- ▶ Techniken
- ▶ Beispiele
- ▶ Ausblick

Grundlagen

- ▶ Was sind Templates?
- ▶ Rahmenbedingungen

Was sind Templates?

- ▶ Methode der generischen Programmierung
- ▶ Compiletime Berechnungen
- ▶ Template typen
 1. Klassentemplates
 2. Funktionstemplates
- ▶ Template Parameter
 1. Typ Parameter
 2. Wert Parameter

Compiletime

- ▶ Nur Konstante/Statiche Parameter erlaubt
- ▶ z.B. keine nicht konstanten Objekte (Wert) oder Referenzen (Wert)
- ▶ Bei komplexeren Algorithmen:
 - Der Compiler rechnet sich eventuell kaputt...
 - Das Programm enthält nur noch Ergebnisse

Template Typen

▶ Klassen Templates

- Erlauben partielle Spezialisierung
- Erlauben Zugriff auf 'Template-Daten', i.e. traits
- Können Teil von Vererbungshierarchien sein

▶ Funktionstemplates

- Erlauben die Generalisierung von Funktionen
- keine partielle Spezialisierung

Template Parameter I

- ▶ integrale Wert Parameter
 - Nur integrale Werte unterliegen keiner Rundung
 - Möglichkeit von Dezimalzahlen z.B.: durch Fixpunkt Arithmetik
- ▶ Typ Parameter
 - Theoretisch jeder Typ
 - Unterscheidung zwischen Pointern, Referenzen und cv-qualifizierten Typen

Template Parameter II

► Und Template Parameter!1

- Template Template-Parameter erlauben templates als Template Parameter :)
- Das Werkzeug für policy based design (dazu später Mehr)

Template Parameter III

- ▶ Default parameter
- ▶ Parameter Deduktion
 - Nicht möglich bei Wert Parametern ;)
 - Komplexe Regel -> siehe Standard
 - verhinderbar durch "one more level indirection"

```
template <class T> struct ID { typedef T result; };  
template <class T> bool getTrue (T x) { return true; }  
template <class T> bool getTrue (typename ID <T>::result x)  
    { return true; }
```

Template-Spezialisierung

- ▶ Erlaubt das Spezialisieren auf bestimmte Parameter-Argumente
- ▶ Das mittel der Wahl um Algorithmen zu implementieren
- ▶ Spezialisierte (optimierte) varianten von Templates möglich
 - z.B. kann `std::copy memcopy(3)` aufrufen
 - Ähnliches verhalten kann durch Überladung erreicht werden
 - Wirklich mächtig in verbindung mit Klassentemplates und partieller Spezialisierung

Techniken

- ▶ Keine Kontrollstrukturen, nur über Spezialisierungen
- ▶ Rein funktionale Programmierung (vergleiche Haskell)
 - **Keine** Seiteneffekte während der Template-Instanziierung
 - Keine Variablen.
 - gleich nochmal: siehe auch Haskell

Techniken/Integral value to Type mapping

- ▶ Erlaubt es zur durch Compilezeit konstanten Integrale Werte zu 'typisieren'
- ▶ Zusätzlich wird der typisierte Wert auch 'gespeichert'
- ▶ Eine Art ad-hoc Loesung um z.B. Funktionen zu Überladen
- ▶ Damit instanziierte Typen sind nur gleich wenn Value gleich ist

```
template <int V>  
struct Int2Type { enum { value = V }; };
```

Techniken/Qualifizierer Entfernen

- ▶ Durch partielle Spezialisierung lassen sich cv-qualifier entfernen

```
template <class T>  
struct UnConst { typedef T result; };
```

```
template <class T>  
struct UnConst <const T> { typedef T result; };
```

- ▶ Analog dazu auch für volatile **und** const volatile

Techniken/Verunpointern

- ▶ Ähnlich wie UnConst <T>:

```
template <class T>
struct UnPointer { typedef T result; };
```

```
template <class T>
struct UnPointer <T*> { typedef T result; };
```

- ▶ Analog dazu auch für Referenzen

Techniken/Type Traits

- ▶ Type traits sind eine Technik, um:
 - Informationen und Eigenschaften zu Typen bereit zustellen
 - Operationen fuer bestimmte Typen zu abstrahieren
 - Entscheidungen basierend auf Typen zur Compilezeit zu treffen

Techniken/Expression Templates

- ▶ Operationen auf Arrays 'horizontalisieren'
- ▶ Sehr Effektiv für grosse Datenmengen
- ▶ Effektive Vermeidung von grossen temporären Objekten
- ▶ Bereits durch vielen Bibliotheken implementiert
- ▶ Funktionsweise am **Beispiel** einer Vektor-Addition $v0 = v1 + v2 + v3$

Techniken/Typelists

- ▶ Wie der Name schon sagt: Listen aus Typen
- ▶ Ermöglicht Operationen auf eine Menge von Typen
 - Anhängen
 - Löschen
 - Suchen
 - was eben so mit Listen geht

Policy Based Design I

- ▶ Klassen werden in Policies 'zerlegt'
- ▶ Eine Policy ist eine kleine Klasse die ein bestimmtes implizites Interface implementiert
- ▶ Eine Host-Klasse erhaelt als Parameter ihre Policies
- ▶ Eine Exponentielle Anzahl von 'Konfigurationen' der Klasse mit ihren Policies ist M"oglich.

STL

- ▶ Eine Sammlung von Standard-Containern, -Algorithmen und Iteratoren
 - Container enthalten Elemente
 - Iteratoren erlauben Zugriff auf Elemente eines Containers
 - Algorithmen Arbeiten mit Iteratoren
 - Iteratoren sind nicht an Container gebunden
- ▶ Basierend auf der Idee von Konzepten
- ▶ keine direkte Unterstuetzung fuer Konzepte in C++
- ▶ wird 'reingehackt' (Concepts sind aber schon fuer C++0x vorgesehen)

Literatur

- ▶ "Die C++ Programmiersprache", Bjarne Stroustrup
- ▶ "Modern C++ Design", *Andrej Alexandrescu*, Addison-Wesley Longman, 2001
- ▶ "Generative Programming", *Krzysztof Czarnecki und Ulrich W. Eisenecker*, Addison-Wesley Longman, 2000
- ▶ Traits: The else-if-then of Types -
<http://erdani.org/publications/traits.html>
- ▶ Expression Templates -
<http://ubiety.uwaterloo.ca/tveldhui/papers/Expression-Templates/exprtmpl.html>

- ▶ `comp.lang.c++ + comp.lang.c++.moderated`
- ▶ try your favourite search engine...

