

Betriebssystemtechnik

Operating System Engineering (OSE)

C++ Templates



1

Motivation

- Produktlinienkomponenten müssen wiederverwendbar sein
 - statisch/dynamisch konfigurierbar
 - vom Anwendungsfall unabhängig (**generisch**) formuliert
- Welche Programmiersprachenmechanismen für die generische Programmierung können genutzt werden?
 - ist GP noch besser geeignet als OO oder AO?
- Analyse am Beispiel von **C++ Templates** [1]
 - heute: technische Grundlagen, Kosten
 - nächstes Mal: Entwurfsansätze



© 2007 Olaf Spinczyk

2

Überblick

- C++ *Template* Grundlagen
 - Klassen- und Funktions-*Templates*
 - Arten von *Template* Parametern
 - Spezialisierung
- Übersetzung und Fallstricke
 - Übersetzungsmodelle
 - *Template* Instanziierung
 - Analyse von *Templates*
- Fortgeschrittene *Templates*
 - *Template* Metaprogramme
 - *Introspection*



© 2007 Olaf Spinczyk

3

Warum *Templates*?

- Häufig werden die selben Algorithmen für verschiedene Datentypen benötigt, z.B. `quicksort()` für `int`, `float`, u.s.w. oder Listen von `int`, `Foo` oder `Bar` Objekten.
- Wie kann der Programmierer damit umgehen, z.B. in C oder Java < 5?
 - Mehrfachimplementierung des Algorithmus
 - Probleme bei der **Wartung**, Wiederholung von Fehlern, Mühe!
 - Gemeinsame Basis
 - fehlende **Typsicherheit** bzw. Typüberprüfung erst zur Laufzeit
 - Präprozessoren (z.B. C Makros)
 - blinde Textersetzung, *Scopes* und *Typen* werden ignoriert
- *Templates* sind ein standardisierter [2] C++ Mechanismus, der alle diese Probleme vermeidet!



© 2007 Olaf Spinczyk

4

Ein erstes Funktions-Template

```
max.h
template <typename T>
inline const T& max (const T &a, const T &b) {
    return a < b ? b : a;
}
```

Template-Parameter (altern. <class T>)

T kann wie ein normaler Typ verwendet werden

- als Template-Parameter kann im Prinzip jeder Typ verwendet werden (keine gemeinsame Basis nötig!)
- Einschränkungen definiert das *Template* implizit, hier:
 - T benötigt operator < (const T&) const
 - die Aufrufparameter von max() müssen den selben Typ haben, damit T „deduziert“ werden kann
 - int i = max(2,3); // ok
 - int k = max(4,4.2); // Fehler



© 2007 Olaf Spinczyk

5

Überladen von Funktions-Templates

```
max.h
template <typename T>
inline const T& max (const T &a, const T &b) {...}

template <typename T>
inline const T& max (const T &a, const T &b, const T &c) {...}

inline const int& max (const int &a, const int &b) {...}
```

```
main.cc
#include "max.h"
int main() {
    max(1,2,3);           // Template mit 3 Argumenten
    max(1.0,2.0);         // max<double> per Deduktion
    max('X','Y');         // max<char> per Deduktion
    max(1,2);             // nicht-Template Variante bevorzugt
    max<>(1,2);            // max<int> per Deduktion
    max<double>(1,2);      // max<double> ohne Deduktion
    max('X',3.14);        // nicht-Template Variante für 2 ints
}
```



© 2007 Olaf Spinczyk

6

Ein erstes Klassen-Template

```
Vector.h
template <typename T>
class Vector {
    T *data;
    int dim;
public:
    // Konstruktor, Copy-Konstruktor(!), ...
    // Zugriffsfunktionen z.B. mit Indexprüfung:
    void set (int index, const T& obj);
    T get (int index) const;
};
```

```
main.cc
#include "Vector.h"
int main() {
    Vector<float> a_vector(3);
    a_vector.set (0, 2.71);
}
```

- Klassen-Templates sind perfekt für Container-Klassen
 - darum gibt es auch die *Standard Template Library* (STL)



© 2007 Olaf Spinczyk

7

Methoden von Klassen-Templates

- Wenn Methoden von Klassen-Templates nicht im Klassenrumpf definiert werden, müssen sie ähnlich wie ein Funktions-Template formuliert werden:

```
vector.h
#include <assert.h>
template <typename T>
void Vector<T>::set (int index, const T& obj) {
    assert (index < dim); // wird nur in der Debug Variante
                          // geprüft
    data[index] = obj;    // erfordert operator = in T
}

template <typename T>
T Vector<T>::get (int index) {
    assert (index < dim); // wird nur in der Debug Variante
                          // geprüft
    return data[index];   // erfordert Copy Konstruktor
}
```



© 2007 Olaf Spinczyk

8

Nicht-Typ Template-Parameter

- neben Typen können auch konstante Ausdrücke als *Template*-Parameter benutzt werden

```
Vector2.h
template <typename T, int DIM>
class Vector2 {
    T data[DIM]; // in jedem Objekt steckt ein Array
public:
    // Konstruktor, Copy-Konstruktor(!), ...
    // Zugriffsfunktionen z.B. mit Indexprüfung:
    void set (int index, const T& obj);
    T get (int index) const;
};
```

- aber **Achtung:**

```
Vector<int> v1(10)
Vector<int> v2(20); // v1 und v2 haben den selben Typ
Vector2<int,10> v2_1; // v2_1 und v2_2 haben
Vector2<int,20> v2_2; // unterschiedliche Typen!
```



Template Template-Parameter

- Templates*, die *Templates* benutzen sollen, können auch *Templates* als Parameter bekommen

```
DataCollector.h
template <template <typename> class Container>
class DataCollector {
    Container<double> collected_data; // Container wird benutzt
public:
    // ...
};
```

Parameter ist ein Template mit einem Parameter

- so wird's benutzt:

```
DataCollector<Vector> dc; // alles OK
DataCollector<Vector2> dc2; // geht nicht!
                          // (Vector2 erwartet 2 Parameter)
```



Default Template-Parameter

- ähnlich wie bei *Default*-Argumenten von Funktionen können auch *Template*-Parameter ein *Default* haben
 - wie bei Funktionen müssen dann nachfolgende Parameter auch ein *Default* haben

```
DataCollector2.h
template <template <typename,int>
        class Container=Vector2, int DIM=32>
class DataCollector2 {
    Container<double, DIM> collected_data; // Container und DIM
public:
    // ...
};
```

zwei Parameter mit Defaults

- so wird's benutzt:

```
DataCollector2<> c1; // auf <> kann man nicht
                   // verzichten
DataCollector2<Vector2> c2; // DIM ist 32 per Default
DataCollector2<FooBar, 10> c3; // Default nicht genutzt
```



Template Spezialisierung

- mit Hilfe der *Template* Spezialisierung können unterschiedliche *Template* Implementierungen in Abhängigkeit von den Parametern gewählt werden
- Explizite Template Spezialisierung**
 - für einen bestimmten Parametersatz, z.B. `Vector<bool>`
- Partielle Template Spezialisierung**
 - für eine Parametermenge, z.B. `Vector<const T*>` mit beliebigem `T`
 - geht **nicht** bei Funktions-Templates
- wenn sowohl eine explizite als auch eine partielle Spezialisierung passen, wird die explizite gewählt
- passt keine Spezialisierung, wird das **primäre Template** genommen
- Template*-Spezialisierung bewirkt eine Fallunterscheidung
 - wichtige Grundlage für *Template*-Metaprogrammierung (S. 29)



Explizite Template Spezialisierung

- kann z.B. zur Optimierung benutzt werden:

```
Vector.h
template <bool>
class Vector<bool> {
    unsigned *data; // bool wird als Bit gespeichert
    int dim;
public:
    // Konstruktor
    Vector(int d) {
        data = new unsigned[1 + (d-1) / (sizeof(unsigned)*8)];
    }
    // Copy-Konstruktor(!), Zugriffsfunktionen, ...
};
```

- mit der Spezialisierung für bool wird für Vector<bool> auf dem Heap nur ein Achtel des Speicherplatzes angefordert



Partielle Template Spezialisierung

- kann z.B. die gleiche Optimierung für Vector2 realisieren:

```
Vector2.h
template <int DIM>
class Vector2<bool, DIM> {
    typedef unsigned storage_t;
    static const unsigned BITS = sizeof(storage_t) * 8;
    storage_t data[1 + (DIM - 1) / BITS];
public:
    // Konstruktor, Copy-Konstruktor(!), ...
    // Zugriffsfunktionen z.B. mit Indexprüfung:
    void set (int index, bool obj) {
        if (obj)
            data[index / BITS] |= (1 << (index % BITS));
        else
            data[index / BITS] &= ~(1 << (index % BITS));
    }
    // ...
};
```



Übersetzung und Fallstricke

- Übersetzungsmodelle
 - Inclusion Model
 - Explicit Instantiation Model
 - Separation Model
- Template Instanziierung
 - Point of Instantiation
 - Instanziierungsketten
 - Code-Generierung
- Analyse von Templates
 - typename
 - template



Übersetzungsmodelle

- normalerweise trennen C++ Programmierer die Schnittstellendefinition (Deklarationen, *.h Datei) von der Implementierung (*.cc Datei)

```
max.h
template <typename T>
const T& max (const T &a, const T &b);
```

```
max.cc
#include "max.h"
template <typename T>
const T& max (const T &a, const T &b) {
    return a < b ? b : a;
}
```

- bei Templates führt das zu Problemen ...
 - max.cc: benötigte Instanzen sind noch unbekannt
 - andere Übersetzungseinheiten: kein Quelltext des Templates



Das Inclusion Model

- die heute gängigste Lösung des Problems ist die **Implementierung** in *.h Datei:

```
max.h
template <typename T>
const T& max (const T &a, const T &b);

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

- max() ist so übrigens **keine** *Inline*-Funktion
- wie werden Duplikate vermieden, wenn zwei Übersetzungseinheiten die gleiche *Template*-Instanz benötigen?
 - Template-Funktionen und Member-Funktionen von Template Klassen werden speziell gebunden. Der Binder vermeidet Duplikate.



Das Explicit Instantiation Model

- wenn z.B. der Code des *Templates* einer Bibliothek nicht offengelegt werden soll, kann man auch die explizite Instanziierung nutzen:

```
max.h
template <typename T>
const T& max (const T &a, const T &b);
```

```
max.cc
#include "max.h"
template <typename T>
const T& max (const T &a, const T &b) {
    return a < b ? b : a;
}
template const int& max (const int &a, const int &b);
template const long& max (const long &a, const long &b);
```

- max() kann so übrigens **nur für int und long** benutzt werden!



Das Separation Model

- ... wird bisher von fast keinem Übersetzer unterstützt:

```
max.h
export template <typename T>
const T& max (const T &a, const T &b);
```

```
max.cc
#include "max.h"
template <typename T>
const T& max (const T &a, const T &b) {
    return a < b ? b : a;
}
```

- inline*-Funktionen können nicht exportiert werden
- ob und wie die C++ Übersetzer in der Zukunft mit dieser Spracheigenschaft umgehen werden, ist noch unklar



Template Instanziierung

```
max.cc
template <typename T>
const T& max (const T &a, const T &b) {
    return a < b ? b : a;
}
```

```
> g++ -c max.cc
> objdump -d -demangle max.o
> |
```

objdump liefert nichts!

- die Template-Definition führt **nicht** zur Code-Generierung
- erst wenn fest steht, welche Template-Instanzen benötigt werden, erzeugt der Übersetzer Code
 - was genau löst die Code-Generierung aus?
 - wo ist der Instanziierungspunkt?
 - wie gut ist der generierte Code?



Code-Erzeugung auf Anforderung

```
tst.cc
template <typename T> struct Array { // ein Template
    T data[T::DIM];
    Array();
};
template <typename T> Array<T>::Array() {} // Konstruktor

struct Foo { enum { DIM=10}; }; // eine Parameterklasse

// diese Zeilen lösen keine Codegenerierung aus!
typedef Array<Foo> FooArray;
FooArray *ptr;
int f(const Array<Foo> &param) { ... }
int f(int) { ... }

// erst für diese Zeilen ist die Codegenerierung nötig
int i = f(3); // Overload Resolution
Array<Foo> valid; // Konstruktor für Objektinstanziierung
template Array<Foo>::Array(); // explizite Instanziierung
```

Der Point Of Instantiation (1)

```
tst.cc
template <typename T> struct Array { // ein Template
    T data[T::DIM]; // erfordert DIM im Parametertypen!
};
struct Foo { enum { DIM=10}; }; // eine Parameterklasse

typedef Array<int> intArray; // OK, obwohl <int> falsch
intArray *ptr; // immernoch kein Problem
int f(const Array<bool>& param) {} // auch OK

// erst hier muss der Übersetzer ein Array<bool> erzeugen
int i = f(3); // FEHLER: 'bool' is not an aggregate type
```

Point Of Instantiation von Array<bool>

- der Point Of Instantiation einer Instanz eines Templates in einer Übersetzungseinheit ist (vereinfacht ausgedrückt) der Punkt, wo der Übersetzer das erste Mal in die Instanz hineinschauen muss

Der Point Of Instantiation (2)

```
tst.cc
// das geht...
struct Outer {
    struct Nested { enum { DIM=2 }; };
    void f() { Array<Nested> obj; } // kein Problem
};

// das geht nicht ...
void g() {
    struct Local { enum { DIM=4 }; };
    Array<Local> obj; // FEHLER: ... uses local type
}
```

Point Of Instantiation von Array<...> wäre eigentlich hier

- der Point Of Instantiation liegt vor dem eventuell umschließenden Funktions-Scope
- Lokale Typen können deshalb nicht als Template-Argumente verwendet werden

Instanziierungsketten

- ... entstehen, wenn Template-Instanzen weitere Instanzen erzeugen
- Fehlermeldungen werden dadurch zu Rätseln ...

```
std::list<std::string> coll;
int main() {
    std::list<std::string>::iterator pos;
    pos = std::find_if(coll.begin(), coll.end(),
        std::bind2nd(std::greater<int>(), "A"));
}
```

```
> g++ -O6 -c tst.cc
/usr/include/c++/3.3.5/bits/stl_algo.h: In function '_InputIter
std::find_if(_InputIter, _InputIter, _Predicate, std::input_iterator_tag)
[with _InputIter = std::_List_iterator<std::string, std::string&,
std::string&, _Predicate = std::bind2nd<std::greater<int>, >]:
/usr/include/c++/3.3.5/bits/stl_algo.h:318: instantiated from '_InputIter std::find_if(_InputIter, _InputIter, _Predicate)
[with _InputIter = std::_List_iterator<std::string, std::string&, std::string&, _Predicate =
std::bind2nd<std::greater<int>, >]:
tst.cc:9: instantiated from here
/usr/include/c++/3.3.5/bits/stl_algo.h:188: error: no match for call to '(
std::bind2nd<std::greater<int>, >) (std::basic_string_char,
std::char_traits_char, std::allocator_char &)'
/usr/include/c++/3.3.5/bits/stl_function.h:395: error: candidates are: typename
_Operation::result_type std::bind2nd<_Operation::operator()(typename
std::greater<int>)
_Operation::first_argument_type&) const [with _Operation =
std::greater<int>]
/usr/include/c++/3.3.5/bits/stl_function.h:401: error:
_Operation::result_type std::bind2nd<_Operation::operator()(typename
_Operation::first_argument_type&) const [with _Operation =
std::greater<int>]
```

Code-Generierung

```
max.cc
const int& max (const int &a, const int &b) {
    return a < b ? b : a;
}

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

template const int& max (const int &a, const int &b);
template const long& max (const long &a, const long &b);
```

- der generierte Code für `max(const int&, const int&)` und `max<int>(...)` ist identisch: **kein Overhead!**
- auf einem 32 Bit x86 mit g++ ist auch `max<int>` und `max<long>` identisch **Vorsicht vor Code-Duplikation!**



Analyse von Templates (1)

- der C++ Standard [2] fordert, dass *Template*-Definitionen bereits soweit wie möglich semantisch analysiert werden
- leider benötigt der Übersetzer dazu Hilfestellungen:

```
S.h
template<typename T>
struct S : X<T>::Base {
    X<T> f() {
        typename X<T>::C * p; // C ist Typ -> Zeigerdeklaration
        X<T>::D *q;           // D ist kein Typ -> Multiplikation!
    }
};
```

- ein `typename`-Präfix ist erforderlich:
 - nur in qualifizierten Namen in Templates
 - nicht in der Liste der Basisklassen und Initialisierer im Konstruktor
 - nur, wenn der Name von einem Template-Parameter abhängt



Analyse von Templates (2)

- ein ähnliches Problem tritt auf, wenn ein *Template* aus einem *Template*-Parameter abhängigen Typ benutzt wird:

```
R.h
template<typename T>
struct R {
    void f() {
        // A ist Template-Typ -> Typumwandlung:
        typename X<T>::template A<1>(0);

        // B ist kein Template -> <(1>0)? :
        X<T>::B<1>(0);
    }
};
```

- die Beachtung von `typename` und `template` macht das Schreiben von Templates kompliziert



Fortgeschrittene Templates

- Template* Metaprogramme
 - Programmierung zur Übersetzungszeit: gezielte Code-Generierung
- Introspection*
 - statische Analyse der vorhandenen Typen
- ... und es gibt noch viel mehr



Template Metaprogramme (1)

- durch rekursive *Template*-Instanziierung und *Template*-Spezialisierung können **zur Übersetzungszeit** Fallunterscheidungen und Schleifen ausgeführt werden:

```
Fac.h
// Fac<I>: instanziiert Fac<I-1> rekursiv
template<int I>
struct Fac {
    enum { RES = I * Fac<I-1>::RES };
};

// Spezialisierung für <0>: terminiert die Meta-Schleife
template<>
struct Fac<0> {
    enum { RES = 1 };
};
```



Template Metaprogramme (2)

- zur Übersetzungszeit ausgeführte Berechnungen kosten zur Laufzeit nichts:

```
main.cc
#include <stdio.h>
#include "fac.h"

int main () {
    printf ("%d\n", Fac<7>::RES); // hier entsteht eine
                                // Instanziierungskette!
}
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    pushl $5040
    pushl $.LC0
    call printf
    leave
    xorl %eax, %eax
    ret
```

- im generierten Assembler-Code steht das Ergebnis (5040) bereits fest!
- mit Metaprogrammen lässt sich praktisch alles berechnen: die Sprache ist **Turing-vollständig**



Introspection

- mittels *Template*-Spezialisierung und dem *Overload-Resolution* Mechanismus kann man zur Übersetzungszeit vieles über vorhandene Typen herausfinden, z.B.:

```
template <typename Base>
struct InheritDetector {
    typedef char (&no)[1];
    typedef char (&yes)[2];
    static yes test(Base *); // Resultatsgröße ist 1
    static no test(...);     // Resultatsgröße ist 2
};

template <typename Derived, typename Base>
struct Inherits {
    typedef Derived *DP;
    // sizeof ermittelt die Größe des Resultatsobjekts
    enum { RET = sizeof(InheritDetector<Base>::test(DP)) ==
                sizeof(InheritDetector<Base>::yes) };
};
```



Zusammenfassung

- C++ Templates bilden das notwendige Handwerkszeug, um mit C++ generisch zu programmieren
- heute existieren zahlreiche Bibliotheken, die darauf basieren
 - die Standard Template Library (STL), Boost, Loki, ...
- der Template Mechanismus ist sehr mächtig
 - Template Spezialisierung (partiell und explizit)
 - erlaubt Template Metaprogrammierung
- Templates sind aber auch sehr komplex und daher nicht leicht zu erstellen
 - Debugging, Fehlermeldungen, Instanziierungsmodelle



Ausblick

- Ansätze zur Trennung der Belange mittels *Templates* und zum Entwurf von Produktlinienkomponenten
 - „*Policy-Based Design*“ (A. Alexandrescu)
 - „*Synthesizing Objects*“ (C. Czarnecki, U. Eisenecker)



Literatur

- [1] D. Vandevor, N.M. Josuttis, *C++ Templates: The Complete Guide*. Addison-Wesley, 2003, ISBN 0-201-73484-2.
- [2] *The C++ Standard – BS ISO/IEC 14882:2003 (Second Edition)*. John Wiley & Sons, Ltd., 2003, ISBN 0-470-84674-7.

