

# Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

22. Dezember 2005

## Nachtrag: const-Funktionen

- Problem: Indirektion durch Member

```
class A {
private:
    int *i;
public:
    A() : i(new int()) { }
    int &get_i() const { // erlaubt: änderbare Referenz zurück
        (*i)++; // erlaubt: Änderung durch Indirektion
        return *i;
    }
}
```

- Compiler nimmt an: `int * const i`

⇒ Der *Zeiger* ist `const`, nicht der `int`, auf den er zeigt  
(das wäre `const int *i`)

- Konvention: `const`-Members geben nichts zurück, mit dem man den Zustand ändern kann

## Nachtrag: const-Funktionen

- 15.12.: `const` member functions
- Idee: Eine `const` Funktion darf keine Daten ändern

```
class A {
private:
    int i;
public:
    int &get_i() const {
        i++; // verboten
        return i; // verboten
    }
}
```

⇒ Bei Übersetzung der `const`-Funktion gelten alle members als `const`

## Heute: Templates

- Templates in C++
  - Orthogonal zu objekt-orientierten Konstrukten
  - Durchgängig in der Standard-Bibliothek
  - Wichtiges Programmier-Paradigma
- Templates und Typen
  - Typvariablen → Einsetzen von Typen
  - Zweite Art von Polymorphie ( $\approx$  Parametrische Polymorphie)
- Templates und Objekt-orientierte Programmierung
  - Ergänzung: Polymorphie jenseits von Subklassen
  - Interaktionen verstehen
  - Verwandt mit Java Generics
- Tricks mit Templates

## Funktions-Templates

```
template<typename T>
T f(int i, const T &x) {
    T y = x + i;
    return y;
}
```

- `template` führt **Template-Parameter** `T` ein
- `T` kann im Rumpf der Definition wie ein normaler Typ verwendet werden
- Funktion `f` funktioniert für "beliebige" Argumente `x`
  - `T=int`
  - `T=S*`
  - ...
- Nur das `+` (und der Copy-Konstruktor) werden benötigt

## Instanziierung und Überladung

- Bei der Instanziierung wird der Rumpf übersetzt
- ⇒ Überladungsauflösung für Funktionen wird durchgeführt
- Nach Ersetzung konkreter Typen für Template-Parameter
  - Suche nach passenden Funktionen und Operatoren
  - Suche nach passenden Konstruktoren, Zuweisung, Destruktoren
- ⇒ Template-Parameter kann spezielle, eigene Operationen "mitbringen"
- ⇒ (Sehr viel Freiraum für elegante & kreative Programmierung)

## Übersetzungsmodell

- Bei Aufruf einer Template-Funktion
  - Compiler berechnet Argumenttypen
  - Compiler berechnet daraus Ersetzung für Template-Parameter
  - Compiler ersetzt Template-Parameter in gesamter Definition⇒ Normale Funktionsdefinition ohne Templates  
⇒ Übersetzung einer normalen Funktion
- Zum Aufruf wird die Template-Funktion **instanziiert**
- Eine Template-Definition mit eingesetzten Parametern heißt auch **Spezialisierung** des Templates

## Beispiel: Aufruf von f

```
class A {
public:
    A(const A&y) : a(y.a) { cout<<"Copy A " <<*this<<endl; }
    ...
    friend A operator+(const A&a, int i);
};
A operator+(const A&a, int i) {
    cout<<a<<"(+A)" <<i<<endl;
    return a;
}
int main() {
    A a(4);
    A b(0);
    b = f(3,a); // Instanz f<A>
}
template<typename T>
T f(int i, const T &x) {
    T y = x + i;
    return y;
}
```

Ausgabe für Aufruf `f(3,a)`

```
Copy A a=4
a=4(+A)3
Copy A a=4
```

---

## Templates und Parameterische Polymorphie

- ML-artige Sprachen (SML, OCaml) bieten [parametrische Polymorphie](#)

```
let f x = (x,x)
```

- Mit let gebundene Bezeichner besitzen [Typschemata](#)

```
f : ∀α.(α → (α × α))
```

- [Typvariable](#)  $\alpha$  kann beliebig ersetzt werden
- Unterschied Templates
  - f wird nur einmal übersetzt und funktioniert für alle Argumente  $\alpha$
  - Rumpf von f kann keine Operationen (außer Bindung/Zuweisung) für speziellen Typ  $\alpha$  benutzen
- C++-Jargon trotzdem: Templates bieten parametrische Polymorphie

---

## Typklassen und Parametrische Polymorphie

- ML-Polymorphie: Keine Operationen für spezielle Instanzen
- Haskell-Erweiterung: [Typklassen](#) definieren benötigte Operationen

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Integer where
  x == y = x `integerEq` y
```

```
elem :: (Eq a) => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) =
  if x == y
  then True
  else elem x ys
```

- elem kann == auf Typ a verwenden, weil es Eq a voraussetzt

---

## Polymorphie und Polymorphie

- Vererbung: *run-time polymorphism*
  - Alle Typen sind fest
  - Spezialfälle (abgeleitete Klassen) überschreiben Funktionen
  - Zuordnung zu konkreten Funktionen durch dynamisch dispatch
  - Erweiterungsmöglichkeit zur Laufzeit
  - Organisation: Hierarchie / directed acyclic graph
- Templates: *compile-time polymorphism / parametric polymorphism*
  - Zuordnung Funktionen zu Typen durch Überladungsauflösung
  - Alle Funktionen werden statisch ausgewählt
  - Compiler muß alle Definitionen kennen
  - Organisation: Beliebig

---

## Explizite Instanziierung

- Normal: Compiler berechnet Instanz aus Argument-Typen
- Alternativ: Explizite Angabe der Spezialisierung in spitzen Klammern
- Manchmal Compiler-Berechnung nicht möglich
  - Template-Parameter kommt nur in der Rückgabe vor

```
template<typename T>
T *create() { T*r= new T(); memset(r,0,sizeof(T)); return r;
...
int *ip = create<int>();
```
  - Template-Parameter werden ohne Konversion ersetzt

```
template<typename T>
T bin_fun(T x, T y) { return x; }
...
bin_fun(3,1.0);           // falsch: T=double oder T=int?
bin_fun<double>(3,1.0); // ok: Spezialisierung T=double
```

---

## Klassen-Templates

```
template<typename T>
class vec {
private:
    T *array;
public:
    vec() : array(new T[10]) { }
};
```

- Klassen können wie Funktionen Templates sein
- Nur explizite Instanziierung
- Klasse wird für jede Spezialisierung neu instanziiert

---

## Nicht-Typ Parameter

- Neben typename auch Wert-Parameter
- Wert-Parameter sind Compilezeit-Konstanten

⇒ Der Compiler kann mit ihnen rechnen

```
template<typename T, int n>
class array {
private:
    T repr[n];
public:
    T &operator[](int i) {
        assert(i>=0 && i<n);
        return repr[i];
    }
};

int main() {
    array<int,10> a;
    a[10] = 0;
}
```

Ergebnis: Assertion failed: (i>=0 && i<n)

---

## Übersetzung "on demand"

```
template<typename T>
class A {
public:
    int f() { return "blunder"; }
    int g() { return "blunder"; }
};

int main() {
    A<int> a;
    a.g();
}
```

- Es werden nur die member functions übersetzt, die auch tatsächlich aufgerufen werden
- f() enthält einen Fehler, wird aber nicht aufgerufen
- Der Fehler in g() wird entdeckt, sobald g() aufgerufen wird

```
templ2.cpp: In member function 'int A<T>::g() [with T = int]':
templ2.cpp:9: instantiated from here
templ2.cpp:5: error: invalid conversion from const char* to int
```

---

## Benutzer-definierte Spezialisierung

- Template-Spezialisierung = Ersetzung von Template-Parametern
- Bisher: Compiler ersetzt bei Benutzung
  - Ersetzung rein mechanisch
  - Es wird der einmal geschriebene Code kompiliert
  - Er muss alle Argumente gleich behandeln
- Spezialfälle könnte man effizienter implementieren
  - vector<bool> braucht 1 Bit pro Element statt einem int
  - vector<T\*> braucht nur eine Implementierung vector<void\*>

⇒ Benutzer kann neuen Code für Spezialfälle angeben

- Prinzip: Benutzer-Spezialisierungen werden Compiler-Spezialisierungen vorgezogen

---

## Beispiel: Benutzer-definierte Spezialisierung

```
template<int n> // übrige Parameter (evtl. template<>)
class array<bool,n> {
private:
    static const int bits_per_int = 8*sizeof(unsigned);
    unsigned repr[n/bits_per_int];
public:
    bool operator[](int i) {
        return (repr[i/bits_per_int] >> (i%bits_per_int))&1;
    }
};

void check_sizes() {
    cout<<"Normal:" <<sizeof(array<int,100><<endl;
    cout<<"Spec:" <<sizeof(array<bool,100><<endl;
}

> ./a.out
Normal:400
Spec:12
```

---

## Programmietechnik: Typen exportieren

- Üblich: Mehrere Typen arbeiten zusammen
    - Graph – Knoten – Kante
    - Liste – Listenknoten
  - Ohne Templates: Alle Typen einfach definieren
  - Mit Templates: Typen sind erst nach Expansion bekannt
- ⇒ Benutzer müsste die Expansion nachvollziehen
- Trick: Exportiere [assoziierte Typen](#) mit typedefs
- ⇒ Compiler berechnet Expansion und stellt Ergebnis bereit
- Spezialfall: Exportiere Template-Parameter

---

## Probleme mit Templates

- Problem: [code bloat](#)
  - Compiler übersetzt jede gefundene Spezialisierung getrennt
  - Logisch äquivalenter Code ist mehrfach im Programm vorhanden
  - Übersetzungszeit wird sehr schnell inakzeptabel
- Fehler in einer Template Definition werden erst bei Benutzung erkannt
  - Sie hängen meist von der speziellen Instanz ab
  - Bei (getesteten) Bibliotheken liegt der Fehler *nur* in der Instanz, die Meldung bezieht sich aber auf die Template-Definition

⇒ Zum Verständnis muss man zur Not die Bibliotheksfunktion lesen
- ⇒ Fehlermeldungen durchbrechen Abstraktionen & Schnittstellen
- ⇒ Fehlermeldungen sind meist schwer verständlich

---

## Beispiel: Iteratoren

```
void print_int_list(list<int> &l) {
    for (list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        list<int>::value_type e = *i;
        cout<<e<<endl;
    }
}

• Besuche alle Elemente einer list<T>
• Konzept: Iterator

- Abstraktion über Zeiger: Zugriff & Zeigerarithmetik
- Assoziierter Typ list<T>::iterator
- list<T>::begin() gibt Iterator auf den Anfang
- list<T>::end() Iterator auf das Ende (die erste Position, die nicht mehr zur Liste gehört)
- i++ schaltet Iterator weiter
- *i ergibt das Listenelement, auf das i zeigt

```

---

## Das Schlüsselwort `typename`

```
template<typename T>
void print_list(list<T> &l) {
    for (typename list<T>::iterator i=l.begin(); i!=l.end(); i++)
        cout<<*i<<endl;
}
```

- Variablendeklaration syntaktisch:  $\langle type \rangle \langle name \rangle$
  - Ein  $\langle type \rangle$  ist nur ein Bezeichner, der vorher als Typ deklariert wurde
  - Syntaktische Unklarheit: Ist `list<T>::iterator` ein Typ?
    - Erst ein konkretes `T` in Spezialisierung von `print_list` bestimmt, welche (mögliche) Spezialisierung von `list<>` verwendet wird
    - Je nachdem wird `iterator` ein Typ sein oder nicht
  - Lösung: `typename` gibt bekannt, dass ein Typ folgt
- ⇒ `typename` wird immer dann verwendet, wenn der Compiler nicht wissen kann, daß ein Typ folgt

---

## Für Liebhaber

- Overload-Resolution: Die IO-Manipulators
- Traits: Dazu wollte ich noch sagen . . .
- Extrem-Syntax mit `typename`s
- Die unreinen: pure virtual functions mit Implementierung

---

## Zusammenfassung C++

- Objekte in C++ sind zunächst Werte
  - Compiler-Unterstützung für benutzer-definierte Wertsemantik
  - Allokation in globalen und lokalen Variablen
  - Effizienzüberlegungen stehen im Vordergrund
- Objekt-orientierte Programmierung
  - Objekte müssen per Referenz oder Zeiger verwaltet werden
  - Virtuelle Funktionen sind Methoden (Virtueller Destruktor!)
  - Problem: Speicherverwaltung muss programmiert werden (Siehe aber `shared_ptr` für reference counting)
- Templates
  - Bieten parametrische Polymorphie, orthogonal zur Vererbungspolymorphie
  - Semantik: Re-Compilierung
    - Spezialisierung & Instanziierung
    - Überladungsauflösung für Operatoren

---

## Wie funktioniert eigentlich `cout<<endl`?

- `cout<<endl` gibt einen Zeilenvorschub aus und leert interne Puffer
- Buffer leeren mit `cout.flush()`
- Klar:
  - `cout` ist ein `ostream`
  - Der `operator<<` ist überladen
- Unklar: Wie kann das Datum `endl` (eine Variable!) dafür sorgen, dass `cout.flush()` aufgerufen wird?

---

## Eine kleine ostream-Klasse

```
class ostream {
public:
    ostream &putc(char c) {
        std::cout<<"Put: "<<(int)c<<std::endl;
        return *this;
    }
    ostream &flush() {
        std::cout<<"Flush"<<std::endl;
        return *this;
    }
    ostream &operator<<(char c) {
        return putc(c);
    }
    ...
};
```

---

## Ergebnis: endl funktioniert

```
int main() {
    ostream out;
    out<<'X'<<endl;
}
```

- endl-Zeiger wird an operator<< übergeben
- Operator ruf endl(out) auf
- Ausgabe:

```
> ./a.out
Put: 88
endl was called!
Put: 10
Flush
```

---

## Erweiterung um endl

```
class ostream {
public:
    ostream &operator<<(ostream &(*f)(ostream&)) {
        return f(*this);
    }
    ...
}
ostream &endl(ostream &out) {
    std::cout<<"endl was called!"<<std::endl;
    out.putc('\n');
    out.flush();
    return out;
}
```

- Hilfsoperator operator<< für Funktionen
    - Ruft die übergebene Funktion auf
    - Übergibt den Stream selbst, damit die Funktion darauf arbeiten kann
  - endl ist eine Funktion von passenden Typ
- ⇒ Compiler übergibt Zeiger auf endl an operator<<

---

## Die iomanip-Bibliothek

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    for (int i=1; i<100; i+=17)
        cout<<setw(3)<<i<<setw(10)<<i*i<<endl;
}
```

Ausgabe: Sauber formatierte Zahlen

```
> ./a.out
1      1
18     324
35     1225
52     2704
69     4761
86     7396
```

---

## Die iomanip-Bibliothek

```
struct _Setw { int n; };
inline _Setw setw(int n) {
    _Setw x;
    x.n = n;
    return x;
}
extern template ostream operator<<(ostream, _Setw);
```

---

## Beispiel: Find

```
template<typename Iterator, typename T>
Iterator find(Iterator first, Iterator last, const T& val)
{
    typename iterator_traits<Iterator>::difference_type
        count = (last - first) / 4;
    ...
}
```

- find erwartet einen beliebigen Iterator  
mit  $O(1)$  Zugriff auf beliebige Elemente einer Sequenz,  $\rightarrow$  random access iterator
- Iterator kann sowohl eine Klasse als auch ein eingebauter Zeiger sein
- Für Iterator-Klasse  $I$ : Zugriff auf  $I::\text{difference\_type}$
- Für Pointer: Bibliothekstyp `ptrdiff_t`

---

## Traits

- Häufig benötigt man Informationen über Typen zur Compilierzeit
    - Anzahl der Bits in der Darstellung
    - Assoziierte Typen
  - Erste Lösung: `const` oder `typedef` in der Definition des Typen
  - Problem: Funktioniert nicht bei
    - Eingebauten Typen
    - Typen aus zugekaufter Bibliothek
  - Zweite Lösung: Traits
    - Definiere leere Template-Klasse, deren Namen die Informationen beschreibt
    - Definiere für jeden Typ eine Spezialisierung mit Informationen
- $\Rightarrow$  Der Compiler sucht die richtige Spezialisierung heraus
- $\Rightarrow$  Funktioniert uniform für Klassen und eingebaute Typen

---

## Beispiel: Iterator Traits

```
template<typename _Iterator>
struct iterator_traits
{
    typedef typename _Iterator::iterator_category iterator_category;
    typedef typename _Iterator::value_type value_type;
    typedef typename _Iterator::difference_type difference_type;
    typedef typename _Iterator::pointer pointer;
    typedef typename _Iterator::reference reference;
};
```

$\rightarrow$  Funktioniert für benutzerdefinierte Iteratoren

```
template<typename _Tp>
struct iterator_traits<_Tp*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp value_type;
    typedef ptrdiff_t difference_type;
    typedef _Tp* pointer;
    typedef _Tp& reference;
};
```

$\rightarrow$  Funktioniert für Zeiger

---

## Typename ist nicht ausreichend

Dank an Arno Fleck

```
template<typename S>
class A {
    template<typename T>
    class B {
        typedef int grail; // find this type!
    };
};
```

- ✓ Klasse A enthält Klasse B
- ✓ Beide sind parametrisiert
- ✓ B enthält einen Typ grail

? Wie kommen wir von außen an grail?

---

## Das erste Hindernis

```
template<typename T>
void arthur2() {
    typedef T::template B<double>::grail grail;
}
```

- error: non-template 'B' used as template
- ⇒ Wir müssen dem Compiler sagen, daß B ein Template ist
- ⇒ Benutze Schlüsselwort `template` analog zu `typename`
- Leider: Noch nicht zufrieden

```
In function 'void arthur2()':
error: expected init-declarator before "grail"
```

- Beobachtung: Woher soll der Compiler wissen, daß `::grail` ein Typ ist?

---

## Der Naive Ansatz

```
template<typename T>
void arthur1() {
    typedef T::B<double>::grail grail;
}
```

- arthur1 ist template-Funktion
  - Sie nimmt an, daß  $T$  so aufgebaut ist wie A, legt sich aber nicht auf diese Wahl fest
  - (Beispiel: Allokator-Klassen in der STL)
  - Sie will an den `grail` (klar!)
- ⇒ Naive Annahme: Der Scope-Operator hilft

```
In function 'void arthur1()':
error: non-template 'B' used as template
```

---

## Gefunden!

```
template<typename T>
void arthur3() {
    typedef typename T::template B<double>::grail grail;
}
```

(Ohne Kommentar)

---

## Pure Virtual Functions mit Implementierung

Dank an Arno Fleck & Roland Weiss

- Pure virtual functions haben keine Implementierung

```
virtual int f() = 0;
```

- Klassen mit pure virtual functions sind abstrakt
- Abstrakte Klassen können nicht instanziiert werden
- Die Überraschung: man kann pure virtual functions mit Implementierung definieren
- Sinn: Abstrakte Klasse mit Grund-Funktionalität:  
Klasse ist nur logisch unvollständig / unbrauchbar

⇒ In Java kann man direkt Klassen als abstract deklarieren, wenn sie gar keine abstract Methoden enthalten

---

## Pure Virtual Functions mit Implementierung

```
class A {
public:
    virtual int f() = 0;
};
int A::f() { // direkt Angabe hinter =0 funktioniert nicht
    return 42;
}
class B : public A {
public:
    int f() { return 53; }
};

int main() {
    B b;
    int i = b.f();
    int j = b.A::f();
    A a; // Fehler: Abstrakte Klasse nicht instanzierbar
}
```

---

## Interessant wären auch noch

- Template Meta Programming: Rechnen mit Typen
  - Spezialisierung ermöglicht Fallunterscheidung⇒ Wie Pattern-Matching in OCaml, nur auf Typen
- Base class injection: Hier erben Väter von ihren Söhnen
  - Template-Parameter sind Typen
  - Klassen können von ihren Template-Parametern erben