



# Generative Programmierung

Andreas Zeller

Lehrstuhl Softwaretechnik  
Universität des Saarlandes, Saarbrücken





# ***Grundidee: Parametrisierung***

---

Die *Abstraktion* ist ein weiteres Grundprinzip der Softwaretechnik.

Seit Beginn der Programmierung wurden Konzepte entwickelt, um mit einem Programm möglichst viele ähnliche Aufgaben lösen zu können.

Durch *Instantiieren* geeigneter Parameter kann ein Programm (eine Funktion, ein Modul. . .) auf verschiedenen Daten arbeiten.

Daten sind klassische Parameter!





# *Parametrisierte Datentypen*

---

Nicht nur Daten, sondern auch *Datentypen* können als Parameter benutzt werden.

Grundidee: Bestandteile des Typs sind variabel – z.B. der Objekttyp bei Containern.

Geht explizit nur in manchen Sprachen:

- Ada – generische Pakete
- C++ – Templates
- Funktionale Sprachen – Polymorphismus





# Generische Stacks in Pizza

---

*Pizza* ist ein Java-Dialekt mit generischen Klassen.

```
public class Stack<E> {  
    // Instanzvariablen und Klassenkonstanten  
    static final int size = 500;  
    E s[] = new E[size];  
    int c = -1;  
  
    // Öffentliche Funktionen  
    public void empty()        { c = -1; }  
    public void push(E x)     { s[++c] = x; }  
    public void pop()         { --c; }  
    public E top()            { return s[c]; }  
    public boolean isempty()  { return c == -1; }  
}
```





## *Generische Stacks in Pizza (2)*

---

Verwendung des Stacks:

```
Stack<int> values = new Stack<int>();  
values.push(42);
```

```
Stack<String> pizza = new Stack<String>();  
pizza.push("Salami");  
pizza.push("Käse");  
pizza.push("Knoblauch");  
pizza.push("Knoblauch");  
pizza.push("Knoblauch");
```

Vorteil: Allgemeinheit, viel bessere Wiederverwendbarkeit

Analog für generische abstrakte Objekte.





# Generische Stacks in Java

---

In 100% reinem Java gibt es keine generischen Datentypen.  
Alternative: eine allgemeine Oberklasse - z.B. Object:

```
public class Stack {
    // Instanzvariablen und Klassenkonstanten
    static final int size = 500;
    Object s[] = new Object[size];
    int c = -1;

    // Öffentliche Funktionen
    public void empty()      { c = -1; }
    public void push(Object x) { s[++c] = x; }
    public void pop()       { --c; }
    public Object top()     { return s[c]; }
    public boolean isempty() { return c == -1; }
}
```





## *Generische Stacks in Java (2)*

---

Nachteil: keine Typsicherheit

- Explizite Typumwandlung nötig
- Mögliche Laufzeit-Fehler bei Typumwandlung:

```
Stack s = new Stack();  
s.push("Hugo");  
string st = (string) s.top(); // OK  
int x = (int)s.top(); // Laufzeitfehler
```

läßt sich problemlos übersetzen, führt aber zu einem Laufzeitfehler.

Bei echten generischen Datentypen würde dies zur Übersetzungszeit erkannt!





# Templates in C++

---

Die Programmiersprache mit den weitestgehenden generischen Möglichkeiten ist C++.

```
template<class T> class Vector { // Template
public:
    explicit Vector(size_t n);    // Konstruktor
    T& operator[] (size_t);      // Feld-Zugriff
}
```

Benutzung:

```
Vector<int> is(100); is[10] = 15;
Vector<double> ds(100); ds[20] = 3.0;
```







# Template-Parameter

---

Als Parameter für Templates sind *Typnamen*, *Aufzählungen* und *Integers* erlaubt:

```
template<class T, int N> class Vector {  
private:  
    T elems[N];  
public:  
    explicit Vector();           // Konstruktor  
    T& operator[] (size_t);     // Feld-Zugriff  
}
```

Benutzung:

```
Vector<int, 100> is; is[10] = 15;
```





# Spezialisierung

---

Oft ist es sinnvoll, die Implementierung eines Containers je nach Typ zu unterscheiden.

```
template<class T> class Vector<bool> {  
    // Vektor für Booleans  
}  
template<class T> class Vector {  
    // Standard-Vektor  
}
```

*Spezialisierungs-Muster* geben Typmuster an:

```
template<class T> class Vector<T *> {  
    // Vektor für Zeiger  
}
```





# ***Standard-Container in C++***

---

In C++ sind zahlreiche Container Bestandteil der Standard-Bibliothek (*standard template library*, STL)

- Sequenzen (vector, list, deque)  
Davon abgeleitet: stack, queue, ...
- Abbildungen (map, multimap)  
Davon abgeleitet: set, multiset, ...

Teilweise spezialisiert für bestimmte Typen





# Alle Container

---

	<code>[]</code>	<code>insert</code>	<code>push_front</code>	<code>push_back</code>	Iteration
<code>vector</code>	$O(1)$	$O(n)+$		$O(1)+$	Ran
<code>list</code>		$O(1)$	$O(1)$	$O(1)$	Bi
<code>deque</code>	$O(1)$	$O(n)$	$O(1)$	$O(1)$	Ran
<code>stack</code>				$O(1)$	
<code>queue</code>			$O(1)$	$O(1)$	
<code>priority_queue</code>			$O(\log n)$	$O(\log n)$	
<code>map</code>	$O(\log n)$	$O(\log n)+$			Bi
<code>multimap</code>		$O(\log n)+$			Bi
<code>set</code>		$O(\log n)+$			Bi
<code>multiset</code>		$O(\log n)+$			Bi
<code>string</code>	$O(1)$	$O(n)+$	$O(n)+$	$O(1)+$	Ran
<code>array</code>	$O(1)$				Ran
<code>valarray</code>	$O(1)$				Ran
<code>bitset</code>	$O(1)$				Ran

“+” bedeutet: von Zeit zu Zeit kommt Extra-Aufwand hinzu





# Sequenzen und Iteratoren

---

Für Sequenzen können *Iteratoren* definiert werden:

```
#include <iostream>
#include <vector>

using namespace std;
int main() {
    vector<long> coll;           // Eine Sammlung
    coll.push_back(9);         // 9 hinzufügen
    coll.push_back(6);         // 6 hinzufügen

    long sum;                  // Die Summe
    vector<long>::iterator itr; // Iterator definieren
    for (itr = coll.begin(); itr != coll.end(); ++itr)
        sum += *itr;
}
```





# *Container und Algorithmen*

---

Die STL stellt zahlreiche Algorithmen bereit, die auf Containern arbeiten.

find gibt das erste passende Element zurück:

```
vector<long>::iterator itr;  
itr = find(coll.begin(), coll.end(), 7);
```





# *Container und Algorithmen*

---

Die STL stellt zahlreiche Algorithmen bereit, die auf Containern arbeiten.

`find` gibt das erste passende Element zurück:

```
vector<long>::iterator itr;  
itr = find(coll.begin(), coll.end(), 7);
```

`find_if` nimmt ein Prädikat:

```
bool less_than_7(long v) { return v < 7; }  
...  
itr = find_if(coll.begin(), coll.end(),  
             less_than_7);
```





## *Container und Algorithmen (2)*

---

for\_each(B, E, F) ruft F() auf für alle Elemente von B bis E:

```
void sum_up(long v) { sum += v; }
```

```
...
```

```
for_each (coll.begin(), coll.end(), sum_up)
```

Insgesamt sind mehr als 70 Algorithmen verfügbar – zum Suchen, Kopieren, Zählen, Sortieren, ...







# *Funktionsobjekte*

---

In den vorangegangenen Beispielen mußten wir die Funktionen und Prädikate (`less_than_7`, `sum_up`) jeweils explizit definieren.

Grund: C++ kennt keine anonymen Funktionen ( $\lambda$ ).

Ansatz: Entsprechende *Funktionsobjekte* definieren





## Funktionsobjekte (2)

---

Wir definieren eine Klasse mit geeignetem ()-Operator:

```
template<class T> class Sum {  
    T res;  
public:  
    Sum(T i = 0): res(i) {}  
    void operator() (T x) { res += x; }  
    T result() const { return res; }  
}
```

Benutzung:

```
Sum<long> s;  
for_each (coll.begin(), coll.end(), s);  
cout << "Summe: " << s.result() << endl;
```





## Funktionsobjekte (3)

---

Die STL stellt passende Funktionsobjekte zur Verfügung:

```
template <class Arg1, class Arg2, class Res>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
}
```

```
template <class T>
struct less: public binary_function<T, T, bool> {
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
}
```

Anwendung: `less<int>(x, y)` ist äquivalent zu `x < y`





# Adapter

---

In der Praxis benötigen wir häufig Funktionsobjekte, deren Argumente teilweise *gebunden* sind – wie etwa `less_than`:

```
template <class T> class less_than {
    T arg2;
public:
    explicit less_than(const T& x): arg2(x) {}
    bool operator() (const T& x) const {
        return less<T>(x, arg2);
    }
}

// ...
itr = find_if(coll.begin(), coll.end(), less_than<int>(7));
```





## Adapter (2)

---

Mit *Adapttern* lassen sich Funktionen auf spezialisiertere Funktionen abbilden („Currying“).

Beispiel:  $f$  sei binäre Funktion

- $\text{bind2nd}(f, y)$  – ruft  $f$  mit  $y$  als zweitem Parameter auf
- $\text{bind1st}(f, x)$  – ruft  $f$  mit  $x$  als erstem Parameter auf

$\text{bind2nd}(\text{less}\langle\text{int}\rangle(), 7)(x)$  ist äquivalent zu  $x < 7$ .

$\text{bind2nd}(\text{less}\langle\text{int}\rangle(), 7)(4)$  ist true.

Weitere Adapter: `mem_fun`, `ptr_fun`, `not1`, ...





## Adapter (3)

---

Beispiele für den Einsatz von Adaptern:

```
// Erstes Element < 7 suchen
binder2nd<less<int> > less_than_7 =
    bind2nd(less<int>(), 7);
itr = find_if(coll.begin(), coll.end(),
             less_than_7);
```





## Adapter (3)

---

Beispiele für den Einsatz von Adaptern:

```
// Erstes Element < 7 suchen  
binder2nd<less<int> > less_than_7 =  
    bind2nd(less<int>(), 7);  
itr = find_if(coll.begin(), coll.end(),  
             less_than_7);
```

```
// Erstes Element == 3 suchen  
itr = find_if(coll.begin(), coll.end(),  
             bind1st(equal_to<int>(), 3));
```

Halleluja!





# *Templates: Vor- und Nachteile*

---

- Abscheuliche Syntax







# *Templates: Vor- und Nachteile*

---

- Abscheuliche Syntax
- + Erhöhte Wiederverwendung (Generizität)





## ***Templates: Vor- und Nachteile***

---

- Abscheuliche Syntax
- + Erhöhte Wiederverwendung (Generizität)
- + Optimierungsmöglichkeiten (Spezialisierung)





# *Metaprogrammierung*

---

Der C++-Compiler muß zur Laufzeit

- Templates instantiieren und
- entscheiden, welche Template-Alternative benutzt wird.

Mit diesen beiden Eigenschaften lassen sich Berechnungen *zur Übersetzungszeit* durchführen.





# Fakultät - dynamisch

---

Wir betrachten diesen klassischen Fakultäts-Code:

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

// ...
cout << "7! = " << factorial(7) << endl;
```

`factorial(7)` wird zur Laufzeit berechnet.





## *Fakultät - statisch*

---

```
template<int n> struct Factorial {  
    enum { RET = Factorial<n - 1>::RET * n };  
}
```

// Spezialisierung

```
template<> struct Factorial<0> {  
    enum { RET = 0 };  
}
```





## Fakultät - statisch

---

```
template<int n> struct Factorial {  
    enum { RET = Factorial<n - 1>::RET * n };  
}
```

// Spezialisierung

```
template<> struct Factorial<0> {  
    enum { RET = 0 };  
}
```

// ...

```
cout << "7! = " << factorial<7>::RET << endl;
```

factorial<7>::RET wird zur *Übersetzungszeit* berechnet!





## *Fibonacci-Zahlen - statisch*

---

```
template<int n> struct Fib {  
    enum { RET = Fib<n - 1>::RET + Fib<n - 2>::RET };  
}  
template<> struct Fib<0> {  
    enum { RET = 0; }  
}  
template<> struct Fib<1> {  
    enum { RET = 1; }  
}
```





## *Fibonacci-Zahlen - statisch*

---

```
template<int n> struct Fib {  
    enum { RET = Fib<n - 1>::RET + Fib<n - 2>::RET };  
}  
template<> struct Fib<0> {  
    enum { RET = 0; }  
}  
template<> struct Fib<1> {  
    enum { RET = 1; }  
}
```

```
// ...
```

```
cout << "fib(10) = " << fib<10>::RET << endl;
```

wird ebenfalls zur Übersetzungszeit berechnet







# Bedingungen

---

```
template<bool cond, class Then, class Else>  
struct IF { typedef Then RET; }
```

```
// Spezialisierung für false  
template<class Then, class Else>  
struct IF<false, Then, Else> { typedef Else RET; }
```





# Bedingungen

---

```
template<bool cond, class Then, class Else>  
struct IF { typedef Then RET; }
```

```
// Spezialisierung für false
```

```
template<class Then, class Else>  
struct IF<false, Then, Else> { typedef Else RET; }
```

```
// ...
```

```
IF<(1 + 2 > 4), short, int>::RET i;
```





# Bedingungen

---

```
template<bool cond, class Then, class Else>  
struct IF { typedef Then RET; }
```

```
// Spezialisierung für false
```

```
template<class Then, class Else>  
struct IF<false, Then, Else> { typedef Else RET; }
```

```
// ...
```

```
IF<(1 + 2 > 4), short, int>::RET i;
```

```
// ...
```

```
const bool have_ssl = true;  
typedef IF<have_ssl, SSLApache, StdApache>::RET Apache;
```





# Bedingungen

---

```
template<bool cond, class Then, class Else>
struct IF { typedef Then RET; }
```

```
// Spezialisierung für false
```

```
template<class Then, class Else>
struct IF<false, Then, Else> { typedef Else RET; }
```

```
// ...
```

```
IF<(1 + 2 > 4), short, int>::RET i;
```

```
// ...
```

```
const bool have_ssl = true;
typedef IF<have_ssl, SSLApache, StdApache>::RET Apache;
```





# Schleifen

---

Wir betrachten eine *iterative* Implementierung zum Berechnen von Fibonacci-Zahlen:

```
int fib(int n)    // n > 0
{
    int i = 1, x = 1, y = 0;
    while (i < n) {
        i = i + 1;
        int x_ = x;
        x = x + y;
        y = x_;
    }
    return x;
}
```

Kann man das so auch *zur Übersetzungszeit* berechnen?





## Schleifen (2)

---

Wir definieren eine *Anweisung* und eine *Bedingung* für die Berechnung von Fibonacci-Zahlen:

```
template<int i_, int x_, int y_> struct FibStat {  
    enum { i = i_, x = x_, y = y_ };  
    typedef FibStat<i + 1, x + y, x> Next;  
}
```

```
template<int n> struct FibCond {  
    template<class Statement> struct Code {  
        enum { RET = Statement::i < n };  
    }  
}
```





## Schleifen (3)

---

Hiermit können wir eine *Schleife* füllen:

```
template<int n> struct Fib {  
    enum { RET = WHILE<FibCond<n>,  
            FibStat<1, 1, 0> >::RET::x };  
}  
...  
cout << "fib(8) = " << Fib<8>::RET << endl;
```





## Schleifen (4)

---

WHILE ist mit Hilfe von IF definiert:

```
template<class Statement> struct STOP {  
    typedef Statement RET;  
}
```

```
template<class Condition, class Statement>  
struct WHILE {  
    typedef typename  
        IF<Condition::template Code<Statement>::RET,  
            WHILE<Condition, typename Statement::Next>,  
            STOP<Statement>  
        >::RET::RET RET;  
}
```

Erkenntnis: Der C++-Compiler ist Turing-universell!







# Anwendungen

---

- Auswahl von Alternativen zur Übersetzungszeit
- Bestimmung von Werten zur Übersetzungszeit
- Gesteuerte Optimierungen (z.B. Loop unrolling)





# Zusammenfassung

---

- *Generische Programmierung* nutzt Parametrisierung, um zur Übersetzungszeit neue Typen zu generieren





# Zusammenfassung

---

- *Generische Programmierung* nutzt Parametrisierung, um zur Übersetzungszeit neue Typen zu generieren
- Templates in C++ ermöglichen
  - erhöhte Wiederverwendung durch Generizität
  - Optimierungsmöglichkeiten für bestimmte Typen und Werte (Spezialisierung)





# Zusammenfassung

---

- *Generische Programmierung* nutzt Parametrisierung, um zur Übersetzungszeit neue Typen zu generieren
- Templates in C++ ermöglichen
  - erhöhte Wiederverwendung durch Generizität
  - Optimierungsmöglichkeiten für bestimmte Typen und Werte (Spezialisierung)
- C++-Templates ermöglichen Metaprogrammierung:
  - Auswahl von Alternativen zur Übersetzungszeit
  - Bestimmen von Werten und Typen zur Übersetzungszeit





# *Literatur*

---

<http://www.research.avayalabs.com/user/wadler/pizza/>

– Alles über Pizza

**Musser et al., STL Tutorial and Reference Guide** – Alles über die STL.

**Stroustrup, The C++ Programming Language** – Alles über C++.

**Czarnecki + Eisenecker, Generative Programming** – Metaprogramming (Kapitel 10).

