K07

Generische Programmierung in C++

1. Templates
2. Funktoren

Templates – Motivation

- Es kommt häufig vor, dass gleiche Funktionen oder Klassen implementiert werden müssen, da sie für verschiedene Typen zur Verfügung stehen sollen:
  - Datenrepräsentation: Matrix, Listen, Bäume...
  - Algorithmen: permutieren, sortieren, ...

- Forderung: Code, der für einen Typ erstellt wurde, sollte für kompatible Typen wiederverwendbar sein
  - \+ Reduktion des Implementierungsaufwandes
  - \+ Bessere Übersicht und Wartung

Templates

- Ein Template (dt.: Schablonen, Vorlagen) ist die Definition einer parametrisierten Funktion oder Klasse
  - Funktions-Template: in der Signatur und/oder im Body wird anstatt eines konkreten Typs ein
    - Parametertyp **T**, oder ein
    - konstanter Parameterwert **v** eines primitiven Typen gesetzt.
  - Klassen-Template: anstatt eines konkreten Typs für ein Attribut einer Klasse oder einen Parameter einer Methode wird ein
    - Parametertyp **T**, oder ein
    - konstanter Parameterwert **v** eines primitiven Typen gesetzt.

Template-Funktion

- Beispiel: **`swap()`**-Funktion
- Verhalten ist für jeden Typ gleich (elementare und komplexe Typen)
- Ineffiziente Variante: schreibe für jeden Typ eine eigene Funktion
- Lösung: definiere eine generische Schablone für die Funktion

`swap()` für Typ **`int`**

```
void swap (int& a, int& b) {
   int tmp = a;
   a = b;
   b = tmp;
}
```

`swap()` für beliebigen Typ **`T`**

```
void swap (T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Deklaration von Template-Funktionen

Die `template <...>` Zeile sagt dem Compiler: *„die folgende Deklaration oder Definition ist als Schablone zu verwenden“*.

Ein Template erwartet als Argumente Platzhalter für Typen: Templateparameter

```
template <class T>
void swap (T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Template-Instanziierung (i)

- Die Umwandlung eines Template in eine konkrete übersetzbare Einheit nennt man Template-Instanziierung
- Das Template (Funktion oder Klasse) wird dabei für einen bestimmten Typ (den Template-Parameter) spezialisiert

```
template <class T>
void swap (T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

- Syntax für explizite Spezialisierung:

```
double d1 = 4.0, d2 = 2.0;
swap<double>(d1, d2);
```

Explizit bedeutet: der Programmierer spezifiziert ausdrücklich, mit welchem Typ die Template-Parameter zu ersetzen sind.

Template-Instanziierung (ii)

Was macht der Compiler bei Template-Instanziierung:

- Instanziierte Templates werden vom Compiler in eine normale Funktion/Klasse umgewandelt.
- Wird **`swap()`** für zwei **`int`**–Werte verwendet, so wird eine Funktion dafür erzeugt

```
void swap(int&, int&)
```

- Wird **`swap()`** für zwei Instanzvariablen vom Typ **`Auto`** verwendet, so wird ebenfalls eine Funktion erzeugt

```
void swap(Auto&, Auto&)
```

- Beim Kompilieren des Templates findet die Typprüfung statt!

Implizite Template-Instanziierung

- Weiteres Beispiel: Berechnung des Minimums von zwei Werten:

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

- Verwendung:

```
double d1 = 4.0, d2 = 2.0;
double m = min(d1, d2);
```

Erlaubt! Typ kann eindeutig aus Programmkontext abgeleitet werden.

```
double d1 = 4.0;
float  f1 = 10.0f;
double m  = min(d1, f1);
```

Das geht nicht! Der Compiler meldet: **„no matching function for call to ´min(double&, float&)´“**

- Alternativen: entweder

```
double m = min<double>(d1, f1);
```

Implizite Typkonvertierung von **f1**

oder

```
double m = min(d1, double(f1));
```

Explizite Typkonvertierung von **f1**

Template-Klassen

- Template-Klassen werden wie Template-Funktionen behandelt

Beispiel für einen **`Stack`**:

Alternative zu **`class`**

```
template <typename T> class Stack {
  private:
    T* inhalt;           // Datenbereich des Stacks
    int index, size;     // Aktueller Index, Grösse des Stacks
  public:
    Stack(int s): index(-1), size(s) {
       inhalt = new T[size]; // Stack als Array implementiert
    }
    void push(T item) { // Ein Element auf den Stack "pushen"
       if (index < (size - 1)) {
          inhalt[++index] = item;
       }
    }
    T top() const {      // Ein Element vom Stack lesen
       if (index >= 0) { return inhalt[index]; }
    }
    void pop() {         // Ein Element auf dem Stack löschen
       if (index >= 0) { --index; }
    }
};
```

Template-Klassen instanziieren

- Verwendung der Template-Klasse **`Stack`**:

```
int main() {
   Stack<int> intStack(100);         // Stack für 100 int
   Stack<double> doubleStack(250); // Stack für 250 double
   Stack<rect> rectStack(50);        // Stack für 50 rect
   intStack.push(7);
   doubleStack.push(3.14);
   rectStack.push(rect(2, 5));
}
```

- **Auch hier gilt:** die entsprechende Klasse wird vom Compiler bei Bedarf aus der Template-Klasse generiert.

Template-Klassen & Polymorphie

```
Stack<Person> personStack(100);  // Nur für Typ Person
                                 // verwendbar
```

- Subklassen von Person können auf diesem **`Stack`** abgelegt werden
  - Dabei werden die Objekte aber in Person konvertiert...
  - wodurch die in Subklassen spezifizierte Funktionalität (und damit auch Polymorphie) verloren geht

Besser: man verwendet einen Zeiger auf die Basisklasse:

```
Stack<Person*> personStack(100);  // Für Person und Subklassen
                                  // verwendbar
```

- Vorteile:
  - Polymorphie funktioniert
  - Subklassen von Person können auf dem **`Stack`** abgelegt werden, ohne deren zusätzliche Information zu verlieren.

Template-Parameter

- ... sind konstante Ausdrücke eines primitiven Datentyps

```
template <class T, int N>
class Buffer {
    T v[N];
  public:
    void clear () {
      for (int i=0; i<N; ++i) v[i] = T(0);
    }
};
```

- Ermöglichen u.A. statische Optimierungen zur Compilezeit (loop-unrolling, ...)

Parameter müssen zur Compilezeit auswertbar und konstant sein!

```
void f (int i) {
  Buffer<char, i> buf; //Fehler
}
```

Templates von Templates

- Templates können als Parameter für andere Templates dienen:

```
complex<float> c1, c2;
swap<complex<float> > (c1, c2);
```


`complex<T>` ist eine Templateklasse der C++ Standard-Bibliothek für Komplexe Zahlen (`#include <complex>`)

Vorsicht: Bei geschachtelten Templates muss man aufeinander folgende '**>**' durch Leerzeichen trennen, da sie sonst als **shift**-Operator (miss)interpretiert werden.

Typprüfung

- Fehler, die durch Template-Parameter entstehen, können nicht vor der ersten Benutzung erkannt werden!

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

```
#include <complex>
/* ... */
complex<double> a, b;
complex<double> c = min<complex<double> >(a, b);
```

Wo / warum beschwert sich der Compiler ?

```
In function ´T min(T, T) [with T = std::complex<double>]´:
...
no match for std::complex<double>& <      std::complex<double>& operator
```

Identische Templates

- Folgende Spezialisierungen führen zu identischen Template-Typen:

```
typedef unsigned int UInt;
Buffer<UInt, 20> buf1;
Buffer<unsigned int, 20> buf2;
```

- Konstante (compile-time) Ausdrücke führen zu identischen Template-Typen:

```
Buffer<int, 5*4> buf1;
Buffer<int, 20> buf2;
```

Template-Spezialisierung

- Manchmal kann es sinnvoll sein, für bestimmte Template-Argumente alternative Implementierungen einzusetzen (z.B. zur Optimierung)

```
template <typename T, int size>
void MyVector::multiply (int d) {
  for (int i = 0; i < size; ++i) {
    data[i] *= T(d);
  }
}
```

Standard (fallback) Implementierung

```
template <>
void MyVector<double,2>::multiply (double d) {
  data[0] *= d;
  data[1] *= d;
}
```

Spezialisierte (optimierte) Implementierung

Template-Spezialisierung

- Die Spezialisierung kann nachträglich hinzugefügt werden, ohne dass der aufrufende Code geändert werden muss

```
MyVector<int, 10> a;
a.multiply(42.0);       // Nutzt Standard Version

MyVector<double, 2> b; // Nutzt automatisch die
b.multiply(42.0);       // spezialisierte Version
```

- Neben kompletten Spezialisierungen (Beispiel auf Folie 16) sind auch partielle Spezialisierungen möglich, z.B.:

```
template <typename T>
void MyVector<T, 2>::multiply (double d) { ... }
```

Trennung von Deklaration und Definition (i)

```
template <class T>
class Stack {
  private:
    T* inhalt;
    int index;
    int size;

  public:
    Stack(int s);
    void push(T item);
    T top() const;
    void pop();
};
```

Header-Datei: **stack.h**

```
#include "stack.h"
using namespace std;
// Achtung: nicht Stack::Stack(int s)!
template <class T>
Stack<T>::Stack(int s):index(-1),size(s) {
    inhalt = new T[size];
}
template <class T>
void Stack<T>::push(T item) {
  if(index<size) {inhalt[++index]=item;}
}
template <class T>
T Stack<T>::top() const {
  if (index>=0) {return inhalt[index];}
}
template <class T>
void Stack<Type>::pop() {
  if (index >= 0) {index--;}
}
```

Definition: **stack.cpp**

Trennung von Deklaration und Definition (ii)

```
#include "stack.cpp"
using namespace std;

int main() {
   Stack<int> intStack(100);
   Stack<double> doubleStack(250);
   intStack.push(7);
   doubleStack.push(3.14);
}
```

- Man beachte **`#include "stack.cpp"`**, nicht **`stack.h`**!
  - Spezifikation allein reicht also nicht
    (egal ob Template-Funktion oder Klasse)

K07

Generische Programmierung in C++

1. Templates
2. **Funktoren**

Funktoren bzw. Funktionsobjekte

- Funktoren sind Funktionen die, dadurch dass sie gleichzeitig ein Objekt sind, einen Zustand bzw. Kontext halten können.

- Beispiel (informell):
  Der *Wieviel-Zeit-bleibt-noch-Funktor*: Berechnet je nach aktuellem Datum (Zustand) und einem Zeitpunkt in der Zukunft (Argument), wie viel Zeit dazwischen noch bleibt.

Funktoren

- Werden in C++ als Klasse (oder Struktur) definiert, die `()`-Operator überlädt.

Beispiel:

```
class divider {
  private: double divisor;
  public:
    divider(double divisor) : divisor(divisor) {}
    double operator()(double divident) { return divident / divisor; }
}
```

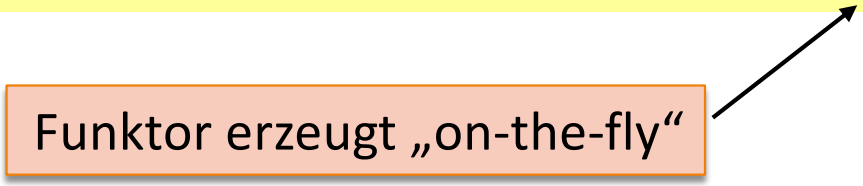
```
divider divideBy2(2.0); // Divisionsfunktor erzeugen u. parametrisieren
divider divideBy3(3.0); //                          - " -

double x = divideBy2(10.0); // Funktor aufrufen
double y = divideBy3(10.0); // Funktor aufrufen
```

Funktoren – Fortsetzung des Beispiels

```
std::vector<double> in;  // Annahme: nichtleerer Vektor
std::vector<double> out; // Zielvektor

// übergebe Divisionsfunktor an std::transform, welche für
// jedes Element von „in“ den Funktor aufruft und das
// Ergebnis in „out“ ablegt
std::transform(
  in.begin(), in.end(), out.begin(), divider(100.0));
```



Funktoren – Zusammenfassung

- Konsequenz aus Objekteigenschaft:
  - Zwangsläufig mit Hilfe von Klassen oder Strukturen definiert
  - Können polymorph sein (virtueller **`operator()`**)
  - Inlining durch Compiler möglich (im Gegensatz zu Funktionszeiger)
  - Können generisch angelegt und bei der Instanziierung parametrisiert bzw. konfiguriert werden

- Konsequenz aus Zustandsbehaftung:
  - Nicht notwendigerweise eine Funktion im mathematischen Sinn

Ok, eine letzte Folie zum Thema C++ ...

```
f(x);  // Was ist hier `f`?
```

Vielleicht:

1. ... eine Funktion, ein Funktionspointer oder eine Funktionsreferenz;
2. ... eine Instanz einer Klasse, also **`operator()`** überladen (Funktor);
3. ... ein Objekt welches implizit konvertierbar zu 1.) oder 2.) ist;
4. ... eine überladene Funktion;
   - in einem der verschiedenen Gültigkeitsbereiche
5. ... der Name eines oder mehrerer Templates;
   - in einem der verschiedenen Gültigkeitsbereiche
6. ... der Name einer oder mehrerer Template-Spezialisierungen;
7. ... eine Kombination der Dinge 1.) bis 7.);
   - z.B. überl. Funktionsname + überl. Template-Name + überl. Template-Spezialisierung
8. ... ein Makro.

Nach einer Folie von Scott Meyers