# Kapitel 17 - Generische Programmierung

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

# Einleitung

- Konzept der generischen Programmierung zielt auf Wiederverwendbarkeit ab
- Umsetzung in C++ mit *Templates* (= Schablonen)
- Ihr kennt sie schon!
  - `std::vector`,
  - `std::unique_ptr`,
- Bisher: nur die Anwendersicht → „gewünschten Typ in Klammern angeben“
  - `std::vector<int> vec; // Enthält nur int`
- Heute: „Öffnen der Motorhaube“ und Erstellen eigener Templates

# Problemstellung

- Aufgabe: Zwei Integer addieren
- Aufgabe: Zwei Fließkommazahlen addieren
- Aufgabe: Zwei 64-bit W[illegible]
- Beobachtung: Kopie für
- Problem
  - Aufwendig
  - Fehleranfällig
  - Wird unübersichtlich

```
1  // Zwei verschiedene Funktionen
2  // notwendig:
3  int add(int sum1, int sum2) {
4      return (sum1 + sum2);
5  }
6
7  double add(double sum1, double sum2) {
       (sum1 + sum2);
```

**Die Lösung: Templates!** 😀

# Templates

- Schlüsselwort `template` kündigt Schablone an
- Abschnitt `<typename T>` beschreibt Schablonenparameter
- `T` als Platzhalter für einen Datentyp

```
1  // Deklaration von Template-Funktionen
2  template <typename T>
3  T add(T sum1, T sum2) {
4      return (sum1 + sum2);
5  }
6
7  // Mehrere verschiedene Typen auch ok
8  template <typename T, typename U>
9  T add_mixed(T sum1, U sum2) {
10     return (sum1 + sum2);
11 }
```

- Platzhalter ersetzt ein Datentyp bei Deklaration: `T eine_variable`
- Konvention: Erster Buchstabe von Platzhaltern wird groß geschrieben

# Beispiel - Template-Funktionen

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std::endl;

// Deklaration von Template-Funktionen
template <typename T>
T add(T sum1, T sum2) {
    return (sum1 + sum2);
}
// Mehrere verschiedene Typen auch ok
template <typename T, typename U>
T add_mixed(T sum1, U sum2) {
    return (sum1 + sum2);
}

int main() {
    cout << "Add double: " << add<double>(2.5, 1.3) << endl;
    cout << "Add int: " << add<int>(4, 3) << endl;
    // Ggf. Rundungsfehler in Implementierung, abhängig von erstem Typ T
```

# Terminologie

- Automatische Umwandlung in C++ → implizite Template-Instanziierung
- Umwandlung eines generischen Typs in einen Konkreten

  → Monomorphisierung (*Monomorphization*)
- Generische Funktionen / Klassen können **neuen** Programmcode erzeugen

  → **Meta**-Funktionen / **Meta**-Klassen

  → **Metaprogrammierung**
- Häufiger Begriff: *Template Metaprogramming* (synonyme Verwendung)
  - *Sehr* beliebt, ein großer Teil der Sprache wird damit implementiert

# Klassenschablonen

- Auch Klassen können eine Schablone sein
- Hier: Datentyp `Pair`
- Zwei Templateparameter

  → Angabe zweier Dat

- Randbemerkung: Gibt es auch in der STL

  → `std::pair` verfügbar

**Bspw. `std::vector<int>`**

```
1  // Einfaches Paaren zweier Klassen
2  template <typename T, typename U>
3  class Pair<T, U> {
4  public:
5      T first;
6      U second;
7      void setFirst(T val) { first = val; }
           d(U val) { second = val; }
11 // Verwendung:
12 class Foo {};
13 class Bar {};
14
15 Pair<Foo, Bar> pair;
```

- Ihr kennt Klassenschablonen schon! Aber woher? 🤔

# Beispiel - Klassenschablonen

cpp Run ▶

```cpp
#include <iostream>

using namespace std;

template <typename T, int NUM>
class Storage {
    T data[NUM];
    int pos;
public:
    Storage() : pos(0) {}
    void put(T val) {
        data[pos] = val;
        pos = (pos + 1) % NUM;
    }
    void print() {
        for (int i = 0; i < NUM; i++) {
            cout << data[i] << ",";
        }
```

# Templates

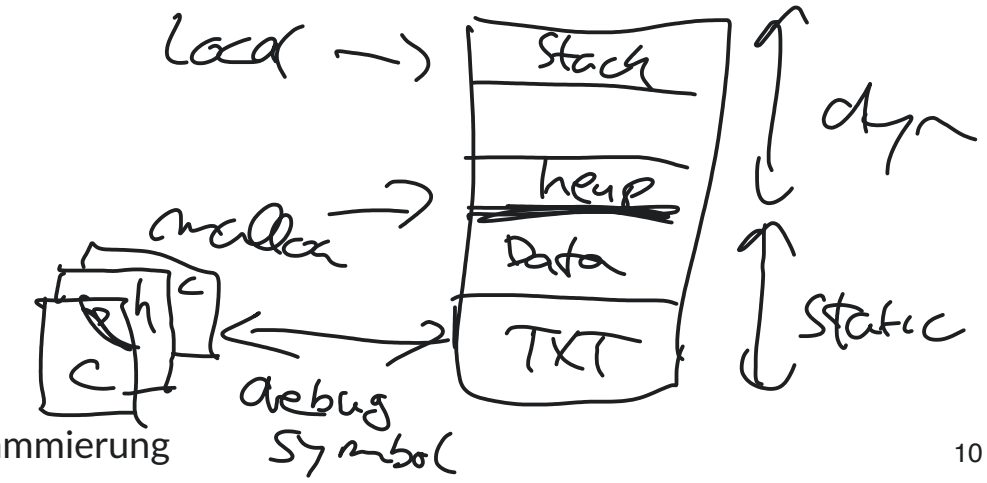
- Während der Übersetzung: Compiler ersetzt Platzhalter
- Compiler findet „unbekannten” Funktionsaufruf
  - „unbekannt” = Neuer Templateparameter
  - **Kopieren** des Template-Codes und Ersetzen der Platzhalter
  - Alle nachfolgenden Aufrufe verwenden dann die erstellte Kopie
- Wichtig: Es wird für **jeden Datentyp** eine eigene Kopie erstellt
- Ergebnis: Automatisch erzeugter Programmschnipsel
  - Ist auch nachher in Binärdatei so vorhanden
  - Für jeden Datentyp wird separater Code erzeugt

# Fallstricke

- Trennung von Programmcode in Header- und Source-Datei nicht möglich

  → Schablonen im Header definieren

- Kann sehr unübersichtlich sein
- Über Tricks ist die Trennung dennoch möglich
  - Deklarierung des Templates im Header, Definition des Template-Codes in separater Datei (wie gehabt)
  - Am **Ende des Headers** `#include` -Anweisung: Fügt Quellcode in den Header ein



# Automatische Ableitung von Parametern

- Compilter **muss** Templateparameter kennen
- Problem: Eine neue Instanz erzeugen?
- Folge: Compiler-Fehler, wenn Doppeldeutigkeiten vorhanden
- Compiler **kann** Parameter automatisch ableiten

cpp Run ▶

```cpp
#include <iostream>

using namespace std;

template <typename T, typename U>
struct Pair {
    T first;
    U second;
    Pair(T f, U u) : first(f), second(u) {}
};

int main() {
    // 0 Typen angegeben. ERROR
    Pair<> p1 = {1, 2};
    cout << p1.first << endl;

    Pair<int, int> p2 = {1, 2}; // Ok
    cout << p2.first << endl;
```

- Empfehlung: Typen immer explizit angeben

# Beispiel - Ableiten von Templateparametern

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std::endl;

template <typename T>
T add(T sum1, T sum2) {
    return (sum1 + sum2);
}

int main() {
    cout << "Add double: " << add(2.5, 1.3) << endl;
    cout << "Add int: " << add(4, 3) << endl;
}
```

# Standardwerte für Schablonenparameter

- Schablonenparameter können Standardwerte haben
- Instanziierung erfolgt mit diesen, ohne explizite Angabe

```
1  template <typename T = int,
2      typename U = double>
3  struct Pair {
4      T first;
5      U second;
6  };
7
8  int main() {
9      Pair p1; // <int, double>
10 }
```

# Herausforderungen mit Schablonen

- Sind nützlicher Teil von C++
- Wichtig: Beachtet einige Punkte
  - Viele Templates → erhöhte Übersetzungszeit (Instanziierung)
  - Kann auch unnötiger Code generiert werden → größeres Binary
  - Fehler in (geschachtelter) Templateprogrammierung sind anspruchsvoll

    → Kryptische Fehlermeldungen beim Kompilieren
- Aber: Eher kleiner Nachteil für flexiblen Programmcode

# Typbeschränkung

- Wunsch: Vorgaben an Schablonenparameter
  - Beispiel: **Addieren** zweier Objekte `Foo` und `Bar` mithilfe von `add()`

  → wenig sinnvoll, sollte nicht erlaubt sein

- Dank C++20: Umsetzung mittels *Constraints* möglich
  - Definieren Beschränkungen (→ Name) für einen Platzhalter
  - Ansonsten: Abbruch während des Kompilierens
- `requires` markiert Einschränkung

```cpp
// add() soll nur Integer bzw.
// float/double akzeptieren
#include <type_traits>

template <typename T>
// T entweder Integer ODER Float
requires std::is_integral_v<T> ||
    std::is_floating_point_v<T>
T add(T sum1, T sum2) {
    return sum1 + sum2;
}
```

# Aufbau von Typbeschränkungen

- Verknüpfung mehrerer Eigenschaften mit && *(logisches UND)* bzw. || *(logisches ODER)*
- Wie bei Mengenlehre: Wird kompiliert, wenn **alle** Eigenschaften oder **eine oder mehr** der Eigenschaften erfüllt werden
- Prüffunktionen geben dabei nur `true` oder `false` zurück

```cpp
// add() soll nur Integer bzw.
// float/double akzeptieren
#include <type_traits>

template <typename T>
// T entweder Integer ODER Float
requires std::is_integral_v<T> ||
    std::is_floating_point_v<T>
T add(T sum1, T sum2) {
    return sum1 + sum2;
}
```

# Beispiel - *Constraints*

cpp Run ▶

```cpp
// add() soll nur Integer bzw.
// float/double akzeptieren
#include <iostream>
#include <type_traits> // für is_integral_v bzw. is_floating_point_v

using std::cout, std::endl;

template <typename T>
// Entweder Integer ODER Float
requires std::is_integral_v<T> || std::is_floating_point_v<T>
T add(T sum1, T sum2) {
    return sum1 + sum2;
}

class Foo {};
int main() {
    cout << add<int>(3,4) << endl;
    cout << add<float>(2.5, 1.3) << endl;
```

# Concepts

- Zusammenfassen von *Constraints*

  → *Concept*

- Bildung eines abstrakten Typen, der alle Eigenschaften erfüllt
- Erspart Schreibarbeit und erlaubt einfache Wiederverwendung

```cpp
#include <type_traits>

// Jetzt als abstrakter Typ
template <typename T>
concept FloatOrInt = std::is_integral_v<T>
    || std::is_floating_point_v<T>;

template <typename T>
// Benutzung identisch
requires FloatOrInt<T>
T add(T sum1, T sum2) {
    return sum1 + sum2;
}
```

- Analog zur objektorientierten Programmierung werden jetzt verschiedene Eigenschaften zu einem gemeinsamen Typ zusammengefasst

# Beispiel - *Concepts*

cpp Run ▶

```cpp
#include <iostream>
#include <type_traits>

using std::cout, std::endl;

// Jetzt als abstrakter Typ
template <typename T>
concept FloatOrInt = std::is_integral_v<T> || std::is_floating_point_v<T>;

template <typename T>
requires FloatOrInt<T> // Benutzung identisch
T add(T sum1, T sum2) {
    return sum1 + sum2;
}

class Foo {};
int main() {
    cout << add<int>(3, 4) << endl;
```

# Typeinschränkung in der STL

- Standardbibliothek bietet eine Reihe von vordefinierten *Constraints* und *Concepts*
- Beispiel: „Klasse X ist abgeleitet von Y“
  - `std::derived_from<X,Y>; // Teil des Headers <concepts>`
- Für vollständige Liste vordefinierter Constraints und Concepts siehe Dokumentation
  - Constraints
  - Concepts-Bibliothek
  - Metaprogramming-Bibliothek

# Resümee Typeneinschränkung

- Fazit: Template-Metaprogramming ist ein zweischneidiges Schwert
- Auf der einen Seite: *Sehr* mächtiger Werkzeugkasten
- Auf der anderen Seite: Wird sehr schnell sehr komplex
- Art und Umfang der Templateprogrammierung hängt vom Projekt ab. Es ist auf jeden Fall sehr nützlich, sie zu kennen.
- Empfehlung: Fangt klein an! Nehmt kleine Beispiele

# Exkurs: *Embedded Template Library*

- Großes Problem: STL auf eingeschränkten Geräten häufig nicht lauffähig
- Simpler Grund: Keine dynamische Speicherallokierung erlaubt
- Betrifft zum Beispiel bereits kennengelernte Speichervergrößerung bei `std::vector` (Verdoppelung des alten Werts)
- Zwei Auswege
  1. Erstellen einer eigenen Allokator-Funktion
  2. Verwenden einer externen Bibliothek: *Embedded Template Library (ETL)*
- Nicht Teil des C++-Standards, reimplementiert aber die Algorithmen und Datenstrukturen mit **statischem** Speicher
- Sehr nützlich, diese zu kennen! 😀

# Exkurs: Statische Polymorphie

- Problem: Virtuelle Methoden werden dynamisch gebunden
  - `vtable` kostet Speicher, Rechenzeit
  - Auf Eingebetteten Systemen ggf. nicht erwünscht
- Statische Polymorphie: Basisklasse als Template
  - Ruft abgeleitete Klasse auf
  - `static_cast` wählt statisch die Kind-Implementierung
  - Keine `vtable` und kein dynamisches Binden mehr

cpp Run ▶

```cpp
#include <iostream>

template <typename T>
struct Shape {
    void draw() const {
        static_cast<const T&>(*this).drawImpl();
    }
};

struct Circle : Shape<Circle> {
    void drawImpl() const {
       std::cout << "Circle\n";
    }
};

struct Square : Shape<Square> {
    void drawImpl() const {
       std::cout << "Square\n";
```

# Zusammenfassung

- Schablonen für Funktionen und Klassen
- Gut: Erlauben wiederverwendbaren Code
  - Einmal schreiben
  - Für verschiedene Datentypen nutzen
  - Reduziert Fehleranfälligkeit
- Aber: Schablonen werden schnell komplex
  - Fehlermeldung teils schwer zu verstehen

Kapitel 17 - Generische Programmierung