8. TEMPLATES

Siegerehrung

Blatt 6

- **Aufgabe 1**
  Adrian Pietsch
- **Aufgabe 2**
  Phan-Long Do (#lines)
- **Aufgabe 3**
  Mark Cule

Inhalt

- **Templatisierung**
  - Templatisierte Funktionen
  - Templatisierte Klassen
  - Spezialisierung
- **λ-Ausdrücke**
  - Syntax
  - Anwendungen

Templates

- **Konzept**: Templates erlauben es, Familien von Klassen oder Funktionen zu definieren.
- Durch Verwendung **generischer Typen** müssen Deklarationen nicht für jeden einzelnen Typ separat implementiert werden.
- Erst der Compiler erzeugt den typspezifischen Code.
- **Vorteile**
  - Kompakter und übersichtlicher Code
  - Leichtere Wartbarkeit
  - Ermöglicht abstrakte Datenstrukturen und Algorithmen, insbesondere in Bibliotheken (`vector`, `stack`, `swap`, `sort` ...)
- GTK Der Compiler optimiert individuell für jeden Typ. Somit gibt es keine Effizienzeinbußen gegenüber typspezifischem Code.

Templatisierte Funktionen

Templatisierte Funktionen

BEISPIEL: Max-Berechnung

- Code für Maximum zweier `ints`

```
int max(int x, int y) { return x > y ? x : y; }
```

- Code für Maximum zweier `floats`

```
float max(float x, float y) { return x > y ? x : y; }
```

- Code für Maximum zweier `chars`

```
char max(char x, char y) { return x > y ? x : y; }
```

Templatisierte Funktionen

BEISPIEL: Max-Berechnung

- Code für Maximum zweier `ints`

```
int max(int x, int y) { return x > y ? x : y; }
```

- Code für Maximum zweier `floats`

```
float max(float x, float y) { return x > y ? x : y; }
```

- Code für Maximum zweier `chars`

```
char max(char x, char y) { return x > y ? x : y; }
```

Funktionsüberladung funktioniert, erfordert aber individuellen Code für jeden Datentyp.

Templatisierte Funktionen

BEISPIEL: Max-Berechnung

- Code für Maximum zweier `ints`

```
int max(int x, int y) { return x > y ? x : y; }
```

- Code für Maximum zweier `floats`

```
float max(float x, float y) { return x > y ? x : y; }
```

- Code für Maximum zweier `chars`

```
char max(char x, char y) { return x > y ? x : y; }
```

Funktionsüberladung funktioniert, erfordert aber individuellen Code für jeden Datentyp.

- Templatecode für Maximum zweier ?

```
T max(T x, T y) { return x > y ? x : y; }
```

Templatisierte Funktionen

BEISPIEL: Max-Berechnung

- Code für Maximum zweier `ints`

```
int max(int x, int y) { return x > y ? x : y; }
```

- Code für Maximum zweier `floats`

```
float max(float x, float y) { return x > y ? x : y; }
```

- Code für Maximum zweier `chars`

```
char max(char x, char y) { return x > y ? x : y; }
```

Funktionsüberladung funktioniert, erfordert aber individuellen Code für jeden Datentyp.

- Templatecode für Maximum zweier ?

```
T max(T x, T y) { return x > y ? x : y; }
```

Wie teilen wir dem Compiler mit, was T bedeutet?

Templatisierte Funktionen

SYNTAX

- Funktionsdeklaration

```
template<class T> return_type fname() {}
template<typename T> return_type fname() {}
```

- Funktionsaufruf

```
int main() {
  fname<T>();
}
```

- Der Compiler generiert erst dann Code, wenn er den jeweiligen Funktionsaufruf sieht.

Templatisierte Funktionen

SYNTAX Beispiel

- Funktionsdeklaration

```
template<typename T>
T max(T x, T y) { return x > y ? x : y; }
```

- Funktionsaufruf

```
int main() {
  max<int>(3, 7);
  max<double>(3.14, 2.7);
  max<char>('h', 'w');
}
```

- Funktioniert für alle Typen, für welche > definiert ist.

Templatisierte Funktionen

SYNTAX Beispiel

- Funktionsdeklaration

```
template<typename T>
T max(T x, T y) { return x > y ? x : y; }
```

- Funktionsaufruf

```
int main() {
  max<int>(3, 7);
  max<double>(3.14, 2.7);
  max<char>('h', 'w');
}
```

- Funktioniert für alle Typen, für welche > definiert ist.
- **GTK** Der Compiler kann (meist) den korrekten Typ raten, wenn man ihn beim Aufruf weglässt.

Templatisierte Funktionen

SYNTAX

- Man kann auch mehrere Typparameter definieren.

```
template<class T, class U> return_type fname() {}
```

- Es ist sogar ein Mix aus Typparametern und Typen möglich.

```
template<typename T, size_t upto>
void print_vector(const vector<T>& v) {
  const size_t stop = min(upto, v.size());
  for (size_t i = 0; i < stop; ++i)
    cout << v[i] << '␣';
  cout << endl;
}
```

- **NGTK** Ab C++20 kann auch `auto` verwendet werden, um kompaktere Funktionstemplates zu deklarieren.

```
auto max(auto x, auto y) { return x > y ? x : y; }
```

Templatisierte Klassen

Templatisierte Klassen

BEISPIEL: Ampel

```
class StringTriple {
 public:
  StringTriple() {}
  StringTriple(string a, string b, string c)
    : f(a), s(b), t(c) {}
  string get_first() { return f; }
  string get_second() { return s; }
  string get_third() { return t; }
 private:
  string f, s, t;
};
```

```
int main() {
  StringTriple traffic_light("red", "yellow", "green");
}
```

Templatisierte Klassen

BEISPIEL: Ampel

```
class StringTriple {
 public:
  StringTriple() {}
  StringTriple(string a, string b, string c)
    : f(a), s(b), t(c) {}
  string get_first() { return f; }
  string get_second() { return s; }
  string get_third() { return t; }
 private:
  string f, s, t;
};
```

```
int main() {
  StringTriple traffic_light("red", "yellow", "green");
}
```

Was müssen wir tun, um die Ampel als Tripel von RGB Farben darzustellen?

Templatisierte Klassen

BEISPIEL: Ampel

```
class StringTriple {
 public:
  StringTriple() {}
  StringTriple(string a, string b, string c)
    : f(a), s(b), t(c) {}
  string get_first() { return f; }
  string get_second() { return s; }
  string get_third() { return t; }
 private:
  string f, s, t;
};
```

```
int main() {
  StringTriple traffic_light("red", "yellow", "green");
}
```

```
class IntegerTriple {
  ...
};

class IntegerTripleTriple {
  ...
};
```

```
int main() {
  IntegerTriple red(255, 0, 0);
  IntegerTriple yellow(255, 255, 0);
  IntegerTriple green(0, 255, 0);

  IntegerTripleTriple traffic_light(red, yellow, green);
}
```

Was müssen wir tun, um die Ampel als Tripel von RGB Farben darzustellen?

Templatisierte Klassen

BEISPIEL: Ampel

```
class StringTriple {
 public:
  StringTriple() {}
  StringTriple(string a, string b, string c)
    : f(a), s(b), t(c) {}
  string get_first() { return f; }
  string get_second() { return s; }
  string get_third() { return t; }
 private:
  string f, s, t;
};
```

```
int main() {
  StringTriple traffic_light("red", "yellow", "green");
}
```

```
class IntegerTriple {
  ...
};

class IntegerTripleTriple {
  ...
};
```

```
int main() {
  IntegerTriple red(255, 0, 0);
  IntegerTriple yellow(255, 255, 0);
  IntegerTriple green(0, 255, 0);

  IntegerTripleTriple traffic_light(red, yellow, green);
}
```

Was müssen wir tun, um die Ampel als Tripel von RGB Farben darzustellen?

Mehrere sehr ähnliche Klassen implementieren.

Templatisierte Klassen

```
template<class T>
class Triple {
 public:
  Triple() {}
  Triple(T a, T b, T c)
    : f(a), s(b), t(c) {}
  T get_first() { return f; }
  T get_second() { return s; }
  T get_third() { return t; }
 private:
  T f, s, t;
};
```

```
int main() {
  Triple<string> traffic_light("red", "yellow", "green");

  Triple<int> red(255, 0, 0);
  Triple<int> yellow(255, 255, 0);
  Triple<int> green(0, 255, 0);

  Triple< Triple<int> > traffic_light2(red, yellow, green);
}
```

Was müssen wir tun, um die Ampel als Tripel von RGB Farben darzustellen?

Mehrere sehr ähnliche Klassen implementieren.

ODER Templates benutzen.

Templatisierte Klassen

SYNTAX

- ```
  template<class T>
  class cname { ... };
  ```
- Es gibt zwei Möglichkeiten zur Instanziierung:
  - Klassenimplementierung in der **Cpp-Datei**
    Vorteil: Linken möglich
    Nachteil: Datei muss konkrete Instanziierungen benennen

    ```
    template class Triple<string>;
    ...
    ```
  - Klassenimplementierung in der **Header-Datei**
    Vorteil: Compiler instanziiert für Aufrufe automatisch, notwendig für Bibliotheken
    Nachteil: Kein Linken
    ```

Templatisierte Klassen

SPEZIALISIERUNG

- Manchmal möchte man für spezielle Parameter andere (effizientere) Implementierungen haben.
- Dies funktioniert durch explizite Angabe der Parameter.

```
template<typename T, size_t rows, size_t columns>
class Matrix {};                    // primary template

template<typename T>
class Matrix<T, 1, 3> {};     // partial specialization

template<>
class Matrix<int, 1, 3> {};   // full specialization
```

- Der Compiler verwendet stets die spezialisierteste passende Klasse zur Instanziierung.
- GTK Bei Klassenüberladung wird zunächst die beste Klasse und erst dann die beste Spezialisierung verwendet.

Templatisierte Klassen

POLYMORPHIE

- Bei einer Instanziierung für eine bestimmte Klasse, können auch Subklassen benutzt werden.

```
stack<Person> pstack;
Child child;
pstack.push(child);
```

Allerdings werden die Subklassen dann konvertiert, und die Polymorphie geht verloren.

- Verwendet man hingegen Zeiger auf die Basisklasse, dann bleibt die Polymorphie erhalten.

```
stack<Person*> pstack;
pstack.push(new Child{});
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL: Kleiner-als

- Funktionsobjekte sind Templates, die Objekte definieren, welche wie eine Funktion aufgerufen werden können.

```
template<typename T>
class Less_than {
  const T val;
 public:
  Less_than(const T& v) : val(v) {}
  bool operator()(const T&x) const { return x < val; }
};

Less_than<int> ltft{42};

int main() { int n = 5; bool b = ltft(n); }
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL: Kleiner-als

- Funktionsobjekte werden oft als Argumente für andere Funktionen und Algorithmen verwendet.

```
template<typename C, typename P>
int count(const C& c, P pred) {
    int cnt = 0;
    for (const auto& x : c)
        if(pred(x)) ++cnt;
    return cnt;
}

void func(const vector<int>& vec){
    cout << count(vec, Less_than<int>{42}) << endl;
}
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL: Kleiner-als

- λs erlauben eine deutlich kompaktere Schreibweise:

```
void func(const vector<int>& vec) {
  cout << count(vec, [](int a) { return a < 42; });
}

void func(const vector<int>& vec, int x) {
  cout << count(vec, [&](int a) { return a < x; });
}
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BASISSYNTAX `[](){};`

- `[]` ist die Capture Klausel
- `()` ist die (optionale) Parameterliste
- `{}` ist die Definition der Funktionalität

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BASISSYNTAX `[](){};`

- `[]` ist die Capture Klausel
- `()` ist die (optionale) Parameterliste
- `{}` ist die Definition der Funktionalität

Variablen einfangen (capture)

```
int a = 1, b = 2, c = 3;
[]{};         // capture no variable
[a]{};        // capture a by value
[&a, b]{};    // capture a by reference and b by value
[=]{};        // capture all variables by value
[&]{};        // capture all variables by reference
[=, &a]{};    // capture all by value but a by reference
[&, a]{};     // capture all by reference but a by value
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

AUFRUF UND ERWEITERTER SYNTAX

- Aufruf an Stelle der Deklaration `[](){}();`

```
int a = 3;
[&](int n) { a += k; }(5);  // a is now 8
```

- Aufruf mit Name (nur lokal gültig)

```
auto lam = []() { cout << "lambda_by_name" << endl; };
lam();  // function call
```

- Mit Rückgabetyp (optional, standardmäßig `auto`)

```
[=]() -> int { return a + b; }
```

- Aufruf als Funktionsargument

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

TYPEN VON LAMBDAS

- Typ 1: Kein Capture, alle Argumenttypen explizit (kein `auto`)

```
[](double x) { return x * x; }
```

  Solche λ sind äquivalent zu Funktionen und können als Funktionspointer übergeben werden.

- Typ 2: Mit Capture und/oder mit `auto` in der Parameterliste

```
[a](double x) { return x * a; }
[](auto x) { return x * x; }
```

  Solche λ müssen mittels Funktionstemplates oder `std::function` übergeben werden.

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL 1: Funktionspointer

```
int square (int x) { return x * x; }

int just_do_it(int (*fnptr)(int), int x) {
  return fnptr(x);
}

int main() {
  int (*fnptr)(int);  // function pointer
  fnptr = square;
  fnptr(2);  // 4
  just_do_it(square, 2);  // 4
  just_do_it([](int x) { return x * x; }, 2);  // 4
}
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL 2: Funktionstemplate

```
template <typename T, typename Function>
void filter(const T& collection, Function f) {
  for (const auto& x : collection)
    if (f(x)) cout << x << '␣';
}
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL 2: Funktionstemplate

```
template <typename T, typename Function>
void filter(const T& collection, Function f) {
  for (const auto& x : collection)
    if (f(x)) cout << x << '␣';
}
```

```
template <typename T>
bool greater_than_five(T x) { return x > 5; }
```

```
vector<int> v = {5, 7, 3, 2, 42};
filter(v, greater_than_five<int>);
```

λ-Ausdrücke

Ein λ(-Ausdruck) ermöglicht es, Funktionsobjekte (anonym) direkt an der Stelle des Aufrufs oder der Übergabe zu definieren.

BEISPIEL 2: Funktionstemplate

```
template <typename T, typename Function>
void filter(const T& collection, Function f) {
  for (const auto& x : collection)
    if (f(x)) cout << x << '␣';
}
```

```
template <typename T>
bool greater_than_five(T x) { return x > 5; }
```

```
vector<int> v = {5, 7, 3, 2, 42};
filter(v, greater_than_five<int>);
```

Mit λ:

```
filter(v, [](int x) { return x > 5; });
```

Anwendungen von Templates und λ

- Templatisierte Container
  `vector<int>`, `vector<double>`, `vector<Cat>` ...
  `pair<int, string>`, `pair<Cat, Dog>` ...
- Templatisierte Funktionen und Operatoren
  `max`, `min`, `swap`, `greater<int>` ...
- Templatisierte Algorithmen

```
#include <algorithm>
int num[] = {3, 2, 4, 1, 8};
sort(num, num + 5, greater<int>);  // 8, 4, 3, 2, 1
```

- λ-Ausdrücke und Operatoren

```
sort(num, num + 5,
  [](int x, int y) { return x > y; });
count_if(num, num + 5,
  [](int x) { return x % 4 == 0; });
```