Programmieren in C++

Templates
generische Programme

Inhalt

- C++ Templates (generische Programme)
- Instanziierung
- Funktions-, Klassen- und Member-Templates
- Templates mit Wertparameter
- Spezialisierung von Templates
- Variadic Templates
- Perfect Forwarding
- Template Templates

C++ Templates (Generics)

- Generische Klassen und Funktionen in C++
  - Template = Schablone = parametrisierbarer Typ
  - typsichere Funktionen und Klassen (Makros machen nur Textersetzung)
  - Quellcode vereinfachen, flexibilisieren und in der Länge reduzieren
  - Implementierung direkt in Header-Dateien
  - erhöht die Kompilationszeit
- Beispiel für überladene Funktion

```
int min( int a, int b ) {
        return ( a < b ) ? a : b; // min for ints
}
char min( char a, char b ) {
        return ( a < b ) ? a : b; // min for chars
}
```

- jetzt generisch

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

```
// Einsatz
int main() {
      int m = min(5, 7);
}
```

Instanziierung

- Instanziierung
  - durch Instanziierung wird aus einem Template eine vollständige Funktion oder Klasse
  - Template wird bei Verwendung implizit instanziiert
  - erfolgt in jeder Kompilationseinheit von Neuem
  - durch explizite Instanziierung kann die Kompilationszeit verkürzt werden

- Beispiel einer generischen Klasse

```
template <typename T>
class A {
    T m_t;
public:
    A(T t) : m_t(t) {}
    void set(T t) { m_t = t; }
};
```

```
// Implizite Instanziierung
int main() {
    A<double> a(3.5);
}
```

Funktions-Template

- Syntax

  template < *TemplateParamListe* > *Funktionsdefinition*

- mit
  - *TemplateParamListe*
    - kommaseparierte Liste von Parametern, welche Typparameter oder Wertparameter sein können; Default-Werte sind erlaubt
    - Beispiele

```
typename TypBezeichner                  // für beliebigen Datentyp (auch primitiv)
class TypBezeichner                     // für beliebige Klasse
template < TemplateParamListe >         // nicht instanziierter generischer Typ
IntegralTypBezeichner Variable          // Wert-Parameter
```

  - *TypBezeichner*
    - ein beliebiger Name, der in der Funktionsdefinition als Datentyp verwendet wird
    - sowohl Grunddatentypen als auch Klassen sind möglich
  - *Funktionsdefinition*
    - übliche Funktionsdefinition, Methode, Konstruktor
    - darf auch ein überladener Operator sein

Funktions-Templates vs. auto

```
auto min1(auto a, auto b) {
   return (a < b) ? a : b;
}

template<typename T0, typename T1>
auto min2(T0 a, T1 b) {
   return (a < b) ? a : b;
}

template<typename T0, typename T1>
auto min3(T0 a, T1 b) -> decltype((a < b) ? a : b) {
   return (a < b) ? a : b;
}

template<typename T>
T min4(T a, T b) {
   return (a < b) ? a : b;
}
```

Klassen-Templates

- Syntax

  template < *TemplateParamListe* > *Klassendefinition*

- mit
  - *TemplateParamListe:* wie bei Funktions-Templates
- Beispiele von generischen Klassen

```
template<typename T>
class Vector {
    T* m_array;
    …
};
template<typename T = char>      // default Zeichentyp ist char
class String {
    T* m_string;
};
```

Templates mit Wert-Parameter

- Wert-Parameter
  - ganzzahlig (ab C++20 auch floating-point möglich)
- Beispiel: statisches Array mit variabler Länge

```
template<typename T, size_t S>
class Array {
    T m_array[S];
public:
    const T& operator[ ](size_t pos) const    { return m_array[pos]; }
    T& operator[ ](size_t pos)                { return m_array[pos]; }
    void print() const {
        cout << '[';
        cout << m_array[0];
        for(size_t i = 1; i < S; i++) cout << ',' << m_array[i];
        cout << ']' << endl;
    }
};
```

Member-Templates

- Idee
  - Funktions-Templates können auch auf Instanzmethoden einer (generischen) Klasse angewendet werden
- Beispiel

```
template<typename T, size_t S> class Array {
   T m_array[S];
public:
   template<typename E> Array(const E& val) {
      for (auto& a : m_array) {
         a = static_cast<T>(val);
      }
   }
   Array() : Array(0) {}
};
```

Spezialisierung von Templates

- Beispiel
  - Minimum von zwei Zahlen oder Zeichen bestimmen
  - bei Zeichen soll die Gross-/Kleinschreibung nicht beachtet werden
- Allgemeinfall

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

- Spezialisierung (muss nach dem Allgemeinfall folgen)

```
template<> char min<char>(char a, char b) {
    a = tolower(a);
    b = tolower(b);
    return ( a < b ) ? a : b;
}
```

Auflistung aller nicht spezialisierten Template-Parameter

Partielle Spezialisierung

- Grundsatz
  - bei mehreren Template-Parametern ist auch nur eine teilweise Spezialisierung erlaubt
  - nur für generische Klassen
- Beispiel

  Allgemeinfall

```
template<typename T, class C> class MyClass {};
```

  partielle Spezialisierung

```
template<class C> class MyClass<char, C> {};
```

  Instanziierungen: Wahl der richtigen Klasse zur Kompilationszeit durch Pattern-Matching (bester und am meisten spezialisierter Match gewinnt)

```
MyClass<int, string> c1;          // Allgemeinfall
MyClass<char, string> c2;         // Spezialisierung
MyClass<char, iostream> c3;       // Spezialisierung
```

Spezialisierung vs. Überladen

- Regel
  - überladene Funktion ist immer ein besserer Match als eine Spezialisierung einer Template-Funktion
- Konsequenz
  - Compiler kann sich kontraintuitiv verhalten
- Beispiel

```
template<typename T> void foo(T x) { cout << "Allgemeinfall" << endl; }
template<> void foo(int* p)      { cout << "Spezialisierung" << endl; }
template<typename T> void foo(T* p)   { cout << "Ueberladen" << endl; }

int i = 5;
foo(i);      // Allgemeinfall
foo(&i);     // Ueberladen
```

Alias Templates

- Idee: Kurzschreibweisen ermöglichen
  - generische Typen können lange Bezeichner erhalten, wenn mehrere generische Parameter verschachtelt werden
  - Beispiel

```
Array<std::vector<std::pair<std::string,int>>, 50> myArray;
```

  - eigene Typen (Alias) definieren mit using

```
template<typename Key, typename Value, size_t N>
using AKV = Array<std::pair<Key, Value>, N>;
```

    - partielle Instanziierung (muss dann verwendet werden, wenn nicht alle Template-Parameter spezifiziert sind)

```
template<typename T> using A50 = Array<T, 50>;
template<size_t N> using Aint = Array<int, N>;
```

    - volle Instanziierung

```
using AV50 = Array<std::vector< uint64_t >, 50>;
using Aint50 = Aint<50>;
```

Variadic Templates

- Templates mit beliebiger Anzahl Argumente
  - Einsatz von Parameter Packs (…)
  - Pattern-Matching zur Kompilationszeit
- Einsatz bei Klassen

```
template<typename… Ts> class C { };
```

Bestimmung der Anzahl Parameter innerhalb der Klasse C

```
size_t types = sizeof…(Ts);
```

- Einsatz bei Funktionen

```
template<typename… Ts> void func(const Ts&… vs) { }
```

Bestimmung der Anzahl Parameter innerhalb der Funktion func

```
size_t params = sizeof…(vs);
```

Variadic Template: Beispiel

```
// zuerst Verankerung der Rekursion
template<typename T> void logging(const T& t) {
    cout << "[0] " << t << endl;
}

// dann Funktion mit beliebiger Anzahl Parameter beliebigen Typs
template<typename First, typename… Rest>
void logging(const First& first, const Rest&… rest) {
    cout << '[' << sizeof...(Rest) << "] ";
    cout << first << ", ";
    logging(rest…); // wird auf alle Elemente des Rests angewendet
}

int main() {
     logging(42, "hallo", 2.3, 'a');
}
```

rest ist ein Parameter Pack

hier wird das Parameter Pack entpackt

Perfect Forwarding

- Generische Rechtsreferenz: Universalreferenz

  template<typename T> void f(T&& x) { ... }

  - Universalreferenz: aus Sicht des Pattern-Matchings ist
    - x ist eine lvalue Referenz (T& x), falls ein lvalue an f übergeben wird
    - x ist eine rvalue Referenz (T&& x), falls ein rvalue an f übergeben wird

  template<typename T> void g(T&& y) { f(y); }

  - innerhalb der Funktion ist y ist ein lvalue (unabhängig davon, was übergeben worden ist)
  - somit wird f mit einem lvalue aufgerufen und daher ist der Parameter y von f eine lvalue Referenz
  - Unschön: an g übergebene Rechtsreferenz wird zu Linksreferenz beim Aufruf von f
- Lösung

  template<typename T> void g(T&& y) { f(std::forward<T>(y)); }

std::forward<T>

- Einsatz
  - wird bei Universalreferenzen (generische Rechtsreferenzen) verwendet
- Nutzen
  - gibt übergebene Linksreferenz als Linksreferenz und übergebene Rechtsreferenz als Rechtsreferenz weiter
- Umsetzung
  - bei einer übergebenen Rechtsreferenz wird std::forward zu std::move
  - wird eine Linksreferenz übergeben, so hat std::forward keine Wirkung
- Beispiel: vereinfachte Implementierung von make_unique

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Perfect Forwarding: Beispiel

```
// zuerst spezialisiertes Funktionstemplate
template<typename T> void logging(T&& t) {
    cout << "[0] " << forward<T>(t) << endl;
}

// dann Funktion mit beliebiger Anzahl Parameter beliebigen Typs
template<typename First, typename… Rest>
void logging(First&& first, Rest&&… rest) {
    cout << '[' << sizeof...(Rest) << "] ";
    cout << forward<First>(first) << ", ";
    logging(forward<Rest>(rest)…);
}

int main() {
     logging(42, "hallo", 2.3, 'a');
}
```

Abhängige Typnamen

- Zweck
  - mit dem Schlüsselwort typename kann dem Compiler mitgeteilt werden, dass ein unbekannter Bezeichner ein Typ ist
  - typename muss verwendet werden, wenn der unbekannte Bezeichner ein vom Template abhängiger qualifizierter Name ist

- Beispiel

```
template<typename T>
struct Extrema {
   using type = typename T::value_type;
   type m_min, m_max;
   Extrema(const T& data)
   : m_min(*min_element(begin(data), end(data)))
   , m_max(*max_element(begin(data), end(data))) {}
};
Extrema<vector<int>> x({ 8, 3, 5, 6, 1, 3 });
```

Template Templates

- Idee
  - ein Template-Parameter darf selber generisch sein, wenn dies entsprechend vordefiniert ist
  - template<template<typname> typename C, typename T> …

- Beispiel

```
template<template<typename, typename> class C, class T, class A>
std::ostream& operator<<(std::ostream& os, const C<T,A>& v) {
   os << '[';
   auto it = v.begin();
   if (it != v.end()) os << *it++;
   for (; it != v.end(); it++) os << ", " << *it;
   os << ']';
   return os;
}
```