Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

8. November 2019

Templates

Inhalt

- Grundlagen
- Spezielle Arten
- Spezialisierung
- Spezialisierung vs. Überladung von Funktions-Templates
- Alias Templates
- Instanziierung
- Template Template
- Variadic Templates

Grundlagen

- Generische Klassen und Funktionen in C++
- Template = Schablone = parametrisierbarer Typ
- Templates: typsicher | Makros: nur Textersetzung
- Quellcode vereinfachen und Länge reduzieren
- erhöht die Compile-Zeit
- keine Source-Files mehr → nur noch Header
- In `template<...>` kann `class` oder `typename` verwendet werden

Best-Practice

- `class` wenn nur komplexe Typen erlaubt (ohne primitive)
- `typename` wenn alle Typen erlaubt

Klassen-Template

Deklaration: `template< ... > Klassendefinition`

```
1  template<typename T = char>
2  class String {
3   public:
4    String() : length_(0) {}
5    explicit String(const T* init)
6        : length_(std::strlen(init))
7        , data_(new T[length_ + 1]) {
8      std::copy(init, init + length_ + 1, data_);
9    }
10  private:
11   size_t length_;
12   T* data_;
13 };
```

Anforderungen an T

- operator=, operator==

Funktions-Template

Deklaration: `template< ... > Funktionsdefinition`

```
1  template<typename T>
2  T Average(T* values, size_t count) {
3    T sum{};  // initialisiert mit ctor oder 0
4    const auto end = values + count;
5    for (auto it = values; it != end; ++it) {
6      sum = sum + *it;
7    }
8    return sum / static_cast<T>(count);
9  }
```

Anforderungen an `T`

- Default-Konstruierbar
- Konvertierbar von `size_t`
- `operator+`, `operator=`, `operator/`

Template-Parameter-Liste

- Kommaseparierte Liste von Parametern
  - Typ- oder Wert-Parameter
  - einfache Beispiele:

```
1 class TypBezeichner
2 typename TypBezeichner, class TypBezeichner
3 class TypBezeichner, size_t Variable
```

- `TypBezeichner`
  - ein beliebiger Name, der in der Funktionsdefinition als Datentyp verwendet wird
  - primitive und/oder Klassen- Typen möglich
- Funktionsdefinition
  - übliche Funktionsdefinition
  - darf auch ein überladener Operator sein

Template mit Wert-Parameter

- Template-Parameter-Liste kann Wert-Parameter enthalten
- Wert-Parameter müssen Integer sein
- Beispiel: `std::array`

```
1  #include <array>
2  #include <string>
3
4  ...
5
6  std::array<int, 3> = {2, 3};
7  // drittes Element ist Default-konstruiert
8
9  std::array<std::string, 2> = {"test", std::string("foo")};
```

Funktions-Template innerhalb Klasse

- auch Methode kann ein Template sein
- gilt auch für Konstruktoren oder Operatoren

```
1  class IntContainer {
2   public:
3    template<typename T>
4    IntContainer(T* init, size_t length) {
5      const auto end = init + length;
6      for (auto it = init; it != end; ++it) {
7        data_.push_back(static_cast<int>(*it));
8      }
9
10  private:
11   std::vector<int> data_;
12 };
```

Spezialisierung von Templates

- nach allgemeinem Fall
- Spezialfall kann komplett anders implementiert sein
- `template<>`-Syntax trotzdem nötig

```
1  // Allgemeinfall
2  template<typename T>
3  T min(T a, T b) {
4    return (a < b) ? a : b;
5  }
6
7  // Spezialfall für char
8  template<>
9  char min<char>(char a, char b) {    // hier überall char verwenden
10   a = std::tolower(a);
11   b = std::tolower(b);
12   return (a < b) ? a : b;
13 }
```

Spezialisierung anwenden

```
1  // Allgemeinfall
2  template<typename T>
3  Klassen-/Funktionsdefinition     // verwendet T
4
5  // Spezialfall für z.B. T=char
6  template<>
7  Klassen-/Funktionsdefinition<char>    // Spezialisierung für T=char
8
9  // Verwendung
10 MyTemplate<int> var_1;       // verwendet Allgemeinfall
11 MyTemplate<char> var_2;      // verwendet Spezialfall
```

- Spezialisierung hat leere Template-Parameter-Liste
- Klassen- oder Funktionsdefinition definiert spezialisierten Typ
- Evaluation zur Compiletime per Pattern-Matching
- Bester und am meisten spezialisierter Match wird gewählt.

partial template specialization

- bei mehreren Template-Parameter ist auch eine teilweise Spezialisierung erlaubt
- Aufruf ändert sich dadurch nicht
- nur für Klassen

```
1  // Allgemeinfall
2  template<typename T, class C>
3  Klassendefinition     // verwendet T und C
4
5  // Spezialfall für z.B. T=char
6  template<class C>
7  Klassendefinition<char, C>    // verwendet char
8                                // C ist aber immer noch frei
9
10 // Verwendung
11 MyTemplate<char, std::vector<char>> var;
```

Spezialisierung vs. Überladung von Funktions-Templates

- Überladene Funktion ist immer besserer Match als Spezialisierung von Template-Funktionen
- Compiler verhält sich unintuitiv
- Why Not Specialize Function Templates?

```
1  template<typename T>    // (a) Allgemeinfall
2  void func(T);
3
4  template<>              // (c) expliziter Spezialfall von (a)
5  void func<int*>(int*);
6
7  template<typename T>    // (b) zweiter Allgemeinfall, überladet (a)
8  void func(T*);
9
10 int* p;      // Verwendet (b). Overload resolution ignoriert
11 func(p);     // Spezialfall und beachtet nur Allgemeinfälle
```

Klassenfunktionen anstatt spezialisierte Funktions-Templates

```
1  // Allgemeinfall
2  template<typename T>
3  struct Min {
4    static T Apply(T a, T b) {
5      ...
6    }
7  };
8
9  // Spezialfall für char
10 template<>
11 struct Min<char> {
12   static char Apply(char a, char b) {
13     ...
14   }
15 };
```

Alias Templates

- generische Typen können lange Bezeichner haben
- Typen wiederverwenden

```
1  std::array<std::vector<std::uint64_t>, 50> data;
```

- mittels `typedef` Typ vor Verwendung definieren
- `using` ist moderner und kann auch nur teilweise definieren

```
1  // mit typedef
2  typedef std::array<std::vector<std::uint64_t>, 50> AV50;
3  AV50 data;
4
5  // mit using
6  template<typename T>
7  using A50 = std::array<T, 50>;
8  using AV50 = A50<std::vector<std::uint64_t>>;
9  AV50 data;
```

Template Instanziierung

- Templates immer nur in Header
- Template wird implizit bei Verwendung instanziiert
- für jede Compilation-Unit separat
- Compiletime optimieren mittels expliziter Instanziierung

```
1  template<typename T>
2  class A { ... };
3
4  // implizite Template Instanziierung
5  A<float> data;
6
7  // deklarieren der expliziten Instanziierung (in .h-Datei)
8  extern template class A<int>;
9  extern template class A<double>;
10
11 // explizite Template Instanziierung für int und double (in .cpp-Datei)
12 template class A<int>;
13 template class A<double>;
```

Template Template

- Ein Template-Parameter darf selber ein Template sein
- Funktioniert nur mit `class` Keyword

```cpp
template<typename T>
struct Cache { ... };

template<typename T>
struct NetworkStore { ... };

template<typename T>
struct MemoryStore { ... };
```

```cpp
template<template<typename> class Store,
         typename T>
struct CachedStore {
  Store<T> store;
  Cache<T> cache;
};

CachedStore<NetworkStore, int> e;
CachedStore<MemoryStore, int> f;
```

```cpp
// ohne Template Template müsste ich den Typ int mehrmals schreiben
CachedStore<NetworkStore<int>, int> e;

// oder könnte einen falschen angeben
CachedStore<MemoryStore<float>, int> f;
```

Variadic Templates

- Template-Parameter-Listen können beliebige Länge haben
- und somit auch die Funktions-Parameter-Liste

```
1  // für Klassen
2  template <typename... Ts>
3  class C {
4  };
5
6  // für Funktionen
7  template <typename... Ts>
8  void fun(const Ts&... vs) {
9  }
10
11 size_t items = sizeof...(Ts);
12 // oder
13 size_t items = sizeof...(vs);
```

Parameter Packs

- Solche Typ-Listen nennen sich Parameter-Packs
- Pattern-Matching für die Evaluation (zur Compiletime)

Definition wenn Operator ...
links von Parameternamen

Entpacken wenn Operator ...
rechts von Parameternamen

```
1  template <typename... Args>
2  auto PerfectLog(const char* format, Args&&... args)                    // Definition
3  {
4    std::string prefix_format("Perfect: ") + format + "\n";
5    return printf(prefix_format.c_str(), std::forward<Args>(args)...);    // Entpacken
6  }
```

Parameter Pack Expansion

Use	Expansion
Ts...	T1, ..., Tn
Ts&&...	T1&&, ..., Tn&&
x<Ts, Y>::z...	x<T1, Y>::z, ..., x<Tn, Y>::z
x<Ts&, Us>...	x<T1&, U1>, ..., x<Tn&, Un>
func(5, vs)...	func(5, v1), ..., func(5, vn)

- Um Elemente einzeln abzuarbeiten muss Rekursion verwendet werden
- Da es sich um Typen handelt werden Templates benötigt
- Dementsprechend kommt Pattern-Matching zum Zug
- Parameter-Pack kann leer sein

Beispiel: Variadic type-safe logging

```cpp
// (a) Basis für einen Parameter
template<typename T>
void LogRecursive(const T& t) {     // besserer Match als (b) mit leerem Rest
  std::cout << t << std::endl;
}

// (b) Overload mit mehr als einem Parameter
template<typename First, typename... Rest>
void LogRecursive(const First& first, const Rest&... rest) {
  std::cout << first << ", ";

  // wird auf alle Elemente des Rest angewendet
  LogRecursive(rest...);
}

LogRecursive(42, "hallo", 2.3, 'a');
// Ruft: b, b, b, a
```