Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

8. November 2019

Template Meta Programming

Inhalt

- Definition
- TMP für Typen
- TMP für Werte
- Diverse Helper
- Type-Traits
- SFINAE
- `std::enable_if`
- Policy based design

Definition

- Template Meta Programming (TMP) bezeichnet Berechnungen zur Compiletime
- Implementieren mittels:
  - öffentliche Template-Klassen (`struct`)
  - Rekursion
  - Vererbung
  - constexpr

Unterscheidung

- TMP für Typen
  - z.B: soll Typ `int` oder `double` verwendet werden
- TMP für Werte
  - z.B: Berechnungen zur Compiletime

TMP für Typen

```cpp
// Allgemeinfall
template<size_t bit_count>
struct TypeForBits {
  using type = uint64_t;    // Design: hat immer type
};

// Spezialfall für 1 Bit
template<>
struct TypeForBits<1> {
  using type = bool;
};

// Verwendung
TypeForBits<6>::type a = 32;
std::cout << a << std::endl;  // 32
TypeForBits<1>::type b = 32;
std::cout << b << std::endl;  // 1
```

TMP für Werte

```
1  template<int a, int b>
2  struct Addition {
3    enum { value = a + b };    // enum-hack
4  };
5
6  constexpr int a = Addition<6, 7>::value;
7  std::cout << a << std::endl;  // 13
```

`enum`-Konstrukt war früher nötig weil:

- ein `static const int value = ...` je nach Verwendung instanziiert werden musste
- Klassen-Variablen nicht in der Deklaration initialisiert werden konnten

TMP für Werte (modern)

```
1  template<int a, int b>
2  struct AdditionConstexpr {
3    static constexpr int value = a + b;
4  };
5
6  // oder
7
8  template<int a, int b>
9  struct AdditionIntegral : std::integral_constant<int, a + b> {};
```

- `constexpr` und expliziter Typ für `value`
- wir möchten die Klasse nicht instanziieren → `static`
- `std::integral_constant` definiert neben `value` auch
  - `value_type`
  - Typkonvertierungs-Operator: `operator value_type()`
  - Funktor-Operator: `operator()`

Keyword constexpr

- Funktioniert je nach Kontext anders
  - Erlaubt Compiletime für Funktionen
  - Forciert Compiletime für Ausdrücke

```
1  constexpr int Subtraction(int a, int b) {
2    return a - b;
3  }
4
5  int SubtractionDynamic(int a, int b) {
6    return a - b;
7  }
8
9  constexpr int a = Subtraction(6, 7);
10 int b = Subtraction(6, dynamic);
11 constexpr int c = Subtraction(6, dynamic);    // Compilefehler
12 constexpr int d = SubtractionDynamic(6, 7);   // Compilefehler
13 int e = SubtractionDynamic(6, dynamic);
```

Assertions für Typen (etc.)

- static_assert für Assertions zur Compiletime
- Für Typen und Compiletime-Werte

```
1  static_assert(Subtraction(6, 7) == -1);
2
3  // Compilefehler: falscher Wert
4  static_assert(Subtraction(6, 7) == 42);
5
6  // Compilefehler: nicht Compiletime
7  static_assert(SubtractionDynamic(6, 7) == -1);
8
9  static_assert(std::is_same<TypeForBits<1>::type, bool>::value);
10 // oder
11 static_assert(std::is_same_v<TypeForBits<1>::type, bool>);
```

Dependent Template Types

- Der Compiler kann nicht entscheiden, ob `::type` ein Typ oder ein `static` Value ist, da er `i` nicht kennt.
- Markieren mit `typename`
- gleiches Problem in vielen Helpern in der Standard-Library
- `using`-Wrapper (`_t`-Suffix) löst das Problem für den Anweder

```
1  template<size_t i>
2  struct BitContainer {
3    using type = std::vector<typename TypeForBits<i>::type>;
4  };
5
6  template<size_t i>
7  using TypeForBits_t = typename TypeForBits<i>::type;
8
9  template<size_t i>
10 struct BitContainerUsing {
11   using type = std::vector<TypeForBits_t<i>>;
12 };
```

Keyword `decltype`

- Herausfinden des Typs einer Variablen oder eines Ausdrucks
- Zur Compiltime
- gleiche Type Inference wie `auto` oder implizite Template-Instanziierung

```
1  struct Account {
2    uint16_t id_;      // wenn Typ ändert, funktioniert Code trotzdem noch korrekt
3    bool valid_;
4  };
5
6  Account CreateAccount(int32_t swisspass_id) {
7    Account a {0, false};
8    using IdType = decltype(a.id_);
9    if (std::numeric_limits<IdType>::max() < swisspass_id) {
10     return a;
11   }
12   a.id_ = static_cast<IdType>(swisspass_id);
13   a.valid_ = true;
14   return a;
15 }
```

Helper `std::declval`

- Erlaubt das Schreiben eines Ausdrucks, welcher eine Instanz einer Klasse ohne Default-Konstruktor benötigt.
- Meist in Kombination mit `decltype`

```
1  struct Default {
2    int Foo() const { return 1; }
3  };
4
5  struct NonDefault {
6    NonDefault(const NonDefault&) {}
7    int Foo() const { return 1; }
8  };
9
10 int main() {
11   decltype(Default().Foo()) n1 = 1;                    // type of n1 is int
12 // decltype(NonDefault().Foo()) n2 = 1;                // error: no default constructor
13   decltype(std::declval<NonDefault>().Foo()) n2 = 1;   // type of n2 is int
14 }
```

Type-Traits

- Evaluierung von Typ Merkmalen
- Header: `type_traits`
- Verwendung z.B. in `static_assert`

```
1  template<typename T>
2  void SpecialFloatHandling(const T& value) {
3    static_assert(std::is_floating_point_v<T>);
4    // ...
5  }
6
7  SpecialFloatHandling(3.14);
8  SpecialFloatHandling(3);  // Compilefehler
```

Substitution failure is not an error (SFINAE)

- Template Substitution bei Instanziierung eines Templates
- Falls kein gültiger Typ für den Template Parameter gefunden werden kann, kann auch das Template nicht instanziiert werden.
- Kein Compilerfehler, aber die Funktion erscheint nicht als verfügbare Überladung.

```cpp
template <typename T>
void func(typename T::SubType) {
  std::cout << "func(T::SubType)" << std::endl;
}

template <typename T>
void func(T) {
  std::cout << "func(T)" << std::endl;
}
```

```cpp
struct WithSubtype {
  using SubType = int;
};

func<WithSubtype>(10);

// funktioniert obwohl
// int kein ::SubType hat
func<int>(10);
```

Type-Traits Hintergrund

- ohne SFINAE, keine Type-Traits möglich

```
1  template<typename T>
2  struct has_type_foo {
3    typedef char true_type[1];
4    typedef char false_type[2];
5
6    template<typename C>
7    static true_type& test(typename C::foo*);
8
9    template<typename>
10   static false_type& test(...);
11
12   enum { value = sizeof(test<T>(nullptr)) == sizeof(true_type) };
13 };
14
15 std::cout << has_type_foo<WithSubtype>::value << std::endl;
16 std::cout << has_type_foo<int>::value << std::endl;
```

Type-Traits Hintergrund (modern)

```
1  using true_type = std::integral_constant<bool, true>;
2  using false_type = std::integral_constant<bool, false>;
3
4  template<typename...> using void_t = void;
5
6  template <typename T, typename = void>
7  struct has_type_foo : std::false_type {};
8
9  template <typename T>
10 struct has_type_foo<T, std::void_t<typename T::foo>> : std::true_type {};
```

std::enable_if

- SFINAE kann Überladungen für bestimmte Typen deaktivieren
- erweitern um Typen ungültig/unbekannt zu machen
- `std::enable_if` kann verwendet werden um beliebige Konstrukte zu deaktivieren

```
1  // Allgemeinfall (hier nur für false)
2  template<bool, typename = void>
3  struct enable_if
4  {};
5
6  // Partial specialization für true
7  template<typename T>
8  struct enable_if<true, T>
9  { using type = T; };
```

Beispiel: `std::enable_if`

```cpp
template<typename T,
         typename std::enable_if_t<std::is_floating_point_v<T>, int> = 0>
void SpecialTypeHandler(T value) {
  std::cout << "Float: " << value << std::endl;
}

template<typename T,
         typename std::enable_if_t<std::is_integral_v<T>, int> = 0>
void SpecialTypeHandler(T value) {
  std::cout << "Int: " << value << std::endl;
}

template<typename T,
         typename std::enable_if_t<std::is_pointer_v<T>, int> = 0>
void SpecialTypeHandler(T value) {
  std::cout << "Pointer: " << value << std::endl;
}
```

Policy based design

- Alternativer Ansatz für Interfaces
- Compiletime anstatt Runtime
- Vererbung und Templates anstatt Dynamic-Binding
- Sinnvolle Verwendung von private inheritance

```
1  class LanguagePolicyEnglish {
2   protected:
3    std::string Message() const {
4      return "Good Day";
5    }
6  };
7
8  class LanguagePolicyGerman {
9   protected:
10   std::string Message() const {
11     return "Guten Tag";
12   }
13 };
```

Policy based design

```
1  template<typename LanguagePolicy>
2  class PolicyUser : private LanguagePolicy {
3    using LanguagePolicy::Message;
4
5   public:
6    void Run() const {  // Behaviour method
7      // policy method
8      std::cout << Message() << std::endl;
9    }
10 };
11
12 PolicyUser<LanguagePolicyEnglish>().Run();  // "Good Day"
13 PolicyUser<LanguagePolicyGerman>().Run();   // "Guten Tag"
```

Buch: Modern C++ Design: Generic Programming and Design Patterns Applied. Andrei Alexandrescu. Addison-Wesley, 2001.