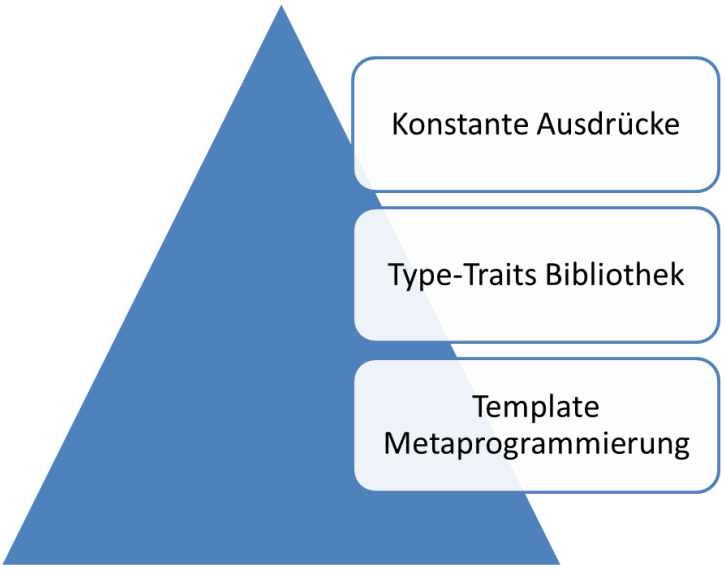
Programmierung zur Compilezeit

Rainer Grimm

Was haben klassische Template Metaprogrammierung, die Funktionen der Type-Traits Bibliothek und konstante Ausdrücke gemein? Sie werden alle zur Compilezeit ausgeführt. Damit vereinen sie höhere Performanz mit erweiterter Funktionalität. Höhere Performanz, da Berechnungen zur Lauf- auf die Compilezeit verlegt werden. Erweiterte Funktionalität, da Programmierung zur Compilezeit den resultierende C++-Sourcecode modifizieren kann. Doch wie funktioniert die ganze Magie?



Template Metaprogrammierung

1994 entdeckte Erwin Unruh Template Metaprogrammierung durch einen Zufall. Sein Programm berechnete die ersten 30 Primzahlen zur Compilezeit. Die Ausgabe der Primzahlen war Teil der Fehlermeldung:

```
01 | Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
02 | Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
03 | Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
04 | Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
05 | Type `enum{}' can't be converted to txpe `D<11>' ("primes.cpp",L2/C25).
06 | Type `enum{}' can't be converted to txpe `D<13>' ("primes.cpp",L2/C25).
07 | Type `enum{}' can't be converted to txpe `D<17>' ("primes.cpp",L2/C25).
08 | Type `enum{}' can't be converted to txpe `D<19>' ("primes.cpp",L2/C25).
09 | Type `enum{}' can't be converted to txpe `D<23>' ("primes.cpp",L2/C25).
10 | Type `enum{}' can't be converted to txpe `D<29>' ("primes.cpp",L2/C25).
```

Wie funktioniert die ganze Magie? Der Compiler instanziiert die Templates und erzeugt den temporären C++-Sourcecode, der zusammen mit dem restlichen Sourcecode übersetzt wird. Zur

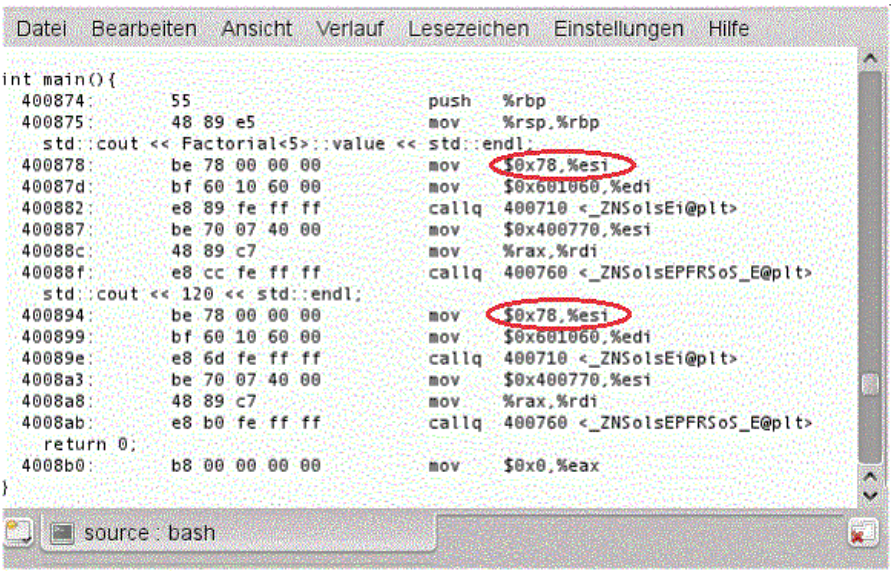
Laufzeit des Programmes steht damit nur der ausführbare Code zur Verfügung.

So führt der Aufruf der Fakultät Funktion `Factorial<5>::value` in dem kleinen Codebeispiel dazu, dass der Wert zur Compilezeit berechnet wird.

```
template <int N>
struct Factorial{
    static int const value= N * Factorial<N-1>::value;
};
template <>
struct Factorial<1>{
    static int const value = 1;
};

std::cout << Factorial<5>::value << std::endl;
std::cout << 120 << std::endl;
```

Schön zeigt die folgende Abbildung, dass der Wert der Faktultät von 5 bereits zur Laufzeit als Konstante vorliegt. Dabei instanziiert der Compiler den Ausdruck `Factorial<5>::value`. Um diesen Wert zu berechnen, benötigt er den Ausdruck `Factorial<4>::value`. Diese Rekursion endet dann, wenn der Wert `Factorial<1>::value` benötigt wird, denn deren Wert ist 1.



Factorial<5>::value zur Compilezeit berechnet

Funktionen wie die `Factorial` Funktion, die zur Compilezeit ausgeführt werden, werden

Metafunktionen genannt. Sie besitzen ein paar interessante Eigenschaften. Neben Ganzzahlen können sie auch Typen und Klassen-Typen als Template-Parameter verwenden. Metafunktionen sind unter der Decke Klassen-Templates. Sie können ihre Daten nicht modifizieren, sondern erzeugen auf Bedarf neue Daten.

Template Metaprogrammierung, die auf Metafunktionen basiert, die zur Compilezeit auf ihren Metadaten agieren, ist eine rein funktionale Subsprache in der imperativen Sprache C++. Diese funktionale Subsprache ist Turing-vollständig [1], und entspricht in ihrer Mächtigkeit der der Programmiersprachen C++, C oder Java.

Template Metaprogrammierung ist die Grundlage für viele Boost-Bibliotheken [2]. Das trifft auch auf die Type-Traits Bibliothek zu, die seit C++11 Teil des C++-Standards ist.

Die Type-Traits Bibliothek

Die Type-Traits Bibliothek erlaubt es, Typabfragen, -vergleiche und –transformationen zur Compilezeit auszuführen. Sie ist eine Anwendung der Template Metaprogrammierung und verfolgt vor allem zwei Ziele: Korrektheit und Optimierung.

Korrektheit

Die Type-Traits Bibliothek erlaubt es in Kombination mit dem `static_assert` Ausdruck, verbindliche Bedingungen an den Sourcecode zu stellen, die zur Compilezeit ausgewertet werden.

Der `gcd`-Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier Ganzzahlen ist viel zu generisch. So lässt er sich zwar mit Fließkommazahlen, Strings und unterschiedlichen Datentypen wie `int(100)` und `long int` (10L) verwenden. Der Compiler quittiert dies aber mit einer sehr wortreichen Fehlermeldung.

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

template<typename T>
T gcd(T a, T b){
  if( b == 0 ) return a;
  else return gcd(b, a % b);
}

int main(){
  std::cout << gcd(100,10) << std::endl;         //  10
  std::cout << gcd(100,33) << std::endl;         //  1
  std::cout << gcd(100,0) << std::endl;          //  100
  std::cout << gcd(3.5,4.0) << std::endl;        // ERROR
  std::cout << gcd("100","10") << std::endl;     // ERROR
  std::cout << gcd(100,10L) << std::endl;        // ERROR
}
```

Durch die Type-Traits Bibliothek lassen sich die Bedingungen an den Algorithmus explizit stellen. Und das alles ohne zusätzliche Kosten für die Laufzeit des Programmes.

```
template<typename T1, typename T2>
typename std::conditional<(sizeof(T1)<sizeof(T2)),T1,T2>::type gcd(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be integral!");
  static_assert(std::is_integral<T2>::value, "T2 should be integral!");
  if( b == 0 )return a;
  else return gcd(b, a % b);
}
```

So prüft der neue `gcd`-Algorithmus, ob die beiden Template-Argumente T1 und T2 Ganzzahlen sind: `std::is_integral<T1>::value.` Als Rückgabetyp für den größten gemeinsamen Teiler gibt der `gcd`-Algorithmus den kleineren der beiden Typen zurück: `std::conditional<(sizeof(T1)<sizeof(T2)),T1,T2>::type.`

Optimierung

Code zu schreiben, der sich beim Übersetzen selbst optimiert, das erlaubt die Type-Traits Bibliothek. So enthält die Standard Template Library (STL) optimierte Versionen der bekannten Algorithmen `std::copy`, `std::fill` oder `std::equal`.
Die Idee ist recht einfach. Immer, wenn die Algorithmen auf Container agieren, deren Elemente hinreichend einfach sind, kommt ein optimierter Algorithmus zum Einsatz. Damit ist es möglich, die Elemente nicht elementweise zu kopieren (`std::copy`), zu füllen (`std::fill`) oder zu vergleichen (`std::equal`), sondern ganze Containerbereiche zu bearbeiten. Dafür kommen die C-Funktionen `memmove`, `memset` und `memcpy` zum Einsatz.

Ob die Containerelemente hinreichend einfach sind, entscheiden zur Compilezeit die Funktionen der Type-Traits Bibliothek.

```
template <typename I, typename T, bool b>
void fill_impl(I first,I last,const T& val,
               const std::integral_constant<bool, b>&){
   while(first != last){
    *first = val;
    ++first;
  }
}

template <typename T>
void fill_impl(T* first, T* last, const T& val, const std::true_type&){
  std::memset(first, val, last-first);
}

template <class I, class T>
inline void fill(I first, I last, const T& val){
  typedef integral_constant<bool,is_trivially_copy_assignable<T>::value
                                 && (sizeof(T) == 1)> boolType;
  fill_impl(first, last, val, boolType());
}
```

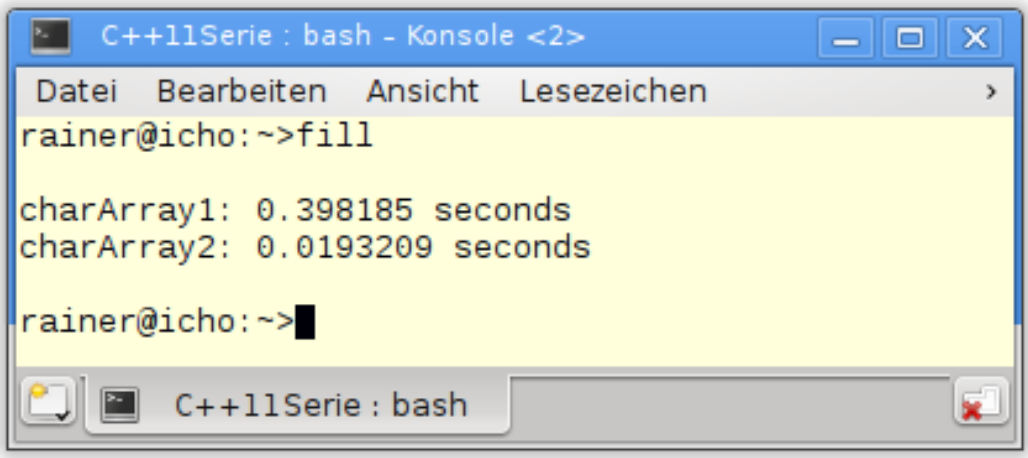
Wird `std::fill` aufgerufen, verwendet der Compiler die Implementierungsfunktion `fill_impl` mit std::memset genau dann, wenn die Elemente des Containers einen vom Compiler automatisch erzeugten Kopierkonstruktor besitzen und ein Byte groß sind: `is_trivially_copy_assignable<T>::value && (sizeof(T) == 1.`

Der Performanzunterschied ist signifikant. So wird in dem Programm ein Array mit 100.000.000 Elementen gefüllt.

```
const int arraySize = 100'000'000;
char charArray1[arraySize]= {0,};
char charArray2[arraySize]= {0,};

int main(){
  auto begin= std::chrono::system_clock::now();
  fill(charArray1, charArray1 + arraySize,1);
  auto last= std::chrono::system_clock::now() - begin;
  cout << "charArray1: " << duration<double>(last).count() << " seconds";
  begin= std::chrono::system_clock::now();
  fill(charArray2, charArray2 + arraySize, static_cast<char>(1));
  last= std::chrono::system_clock::now() - begin;
  cout << "charArray2: " << duration<double>(last).count() << " seconds";
}
```

Während die Elemente des `charArray1` mit der Zahl 1 initialisiert werden, die in der Regel 4 oder 8 Byte große ist, wird `charArray2` mit einer `char` initialisiert: `static_cast<char>(1)`.



Die performante Version des Algorithmus ist um den Faktor 20 schneller.

Konstante Ausdrücke

Konstante Ausdrücke können zur Compilezeit evaluiert und im ROM gespeichert werden. Sie geben dem Compiler tiefen Einblick in den Sourcecode und sind implizit threadsicher.

Konstante Ausdrücke gibt es als Variablen, benutzerdefinierte Typen und Funktionen. Im Gegensatz zu Template Metaprogrammierung erlauben sie das Programmieren zur Compilezeit im imperativen Stil.

Damit konstante Ausdrücke zur Compilezeit evaluiert werden können, gilt es ein paar Regeln zu beachten.

- Variablen sind implizit `const` und müssen durch konstante Ausdrücke initialisiert werden: `constexpr double pi= 3.14;`
- Benutzerdefinierte Typen können keine virtuelle Basisklasse besitzen. Ihr Konstruktor muss vom Compiler erzeugt werden und selbst ein konstanter Ausdruck sein. Methoden, die Objekte von benutzerdefinierten Typen verwenden, müssen konstante Ausdrücke und können nicht virtuell sein.
- Funktionen können keine statische und `thread_local` Variablen enthalten.

Die `gcd`-Funktion ist ein konstanter Ausdruck. Sind ihre Argumente auch konstante Ausdrücke, kann sie zur Compilezeit evaluiert werden. Genau das zeigt das folgende Programm.

```
#include <iostream>

constexpr auto gcd(int a, int b){
  while (b != 0){
    auto t= b;
    b= a % b;
    a= t;
  }
  return a;
}

int main(){
  constexpr int i= gcd(11,121);
  std::cout << i << std::endl;            // 11
  constexpr int j= gcd(100,1000);
  std::cout << j << std::endl;            // 100
}
```

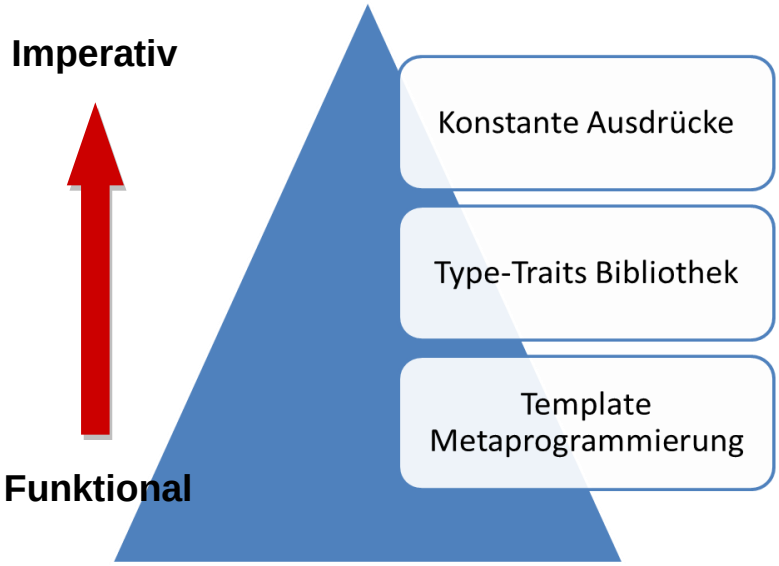
Ein tiefer Blick in den Objektcode zeigt, dass die Ergebnisse der `gcd`-Aufrufe bereits zur Compilezeit vorliegen:

```
  mov $0xb,%esi
  mov $0x601080,%edi
  ...
  mov $0x64,%esi
  mov $0x601080,%edi
  ...
```

Konstante Ausdrücke versus Template Metaprogrammierung

constexpr-Funktionen unterscheiden sich im Detail deutlich von Metafunktionen, die bei der Template Metaprogrammierung zum Einsatz kommen. So können konstante Ausdrücke zur Compile- und Laufzeit ausgeführt werden, Sprung- und Iterationsanweisungen sowie auch bedingte Anweisungen enthalten. Darüber hinaus unterstützen constexpr-Funktionen das Verändern von Daten und das Arbeiten mit Fließkommazahlen.

Programmierung zur Compilezeit als mächtiges Werkzeug



Template Metaprogrammierung, die Type-Traits Bibliothek und konstante Ausdrücke haben gemein, dass sie zu Compilezeit ausgeführt werden. Während Template Metaprogrammierung aber ein tiefes Verständnis von rein funktionaler Programmierung voraussetzt, ist der Einsatz der neuen C++11 Type-Traits Bibliothek deutlich intuitiver für den imperativ geschulten Programmierer. Nur die Namenskonventionen erinnern an Template Metaprogrammierung. Konstante Ausdrücke mit dem C++14-Standard erlauben direkt das Programmieren im imperativen Stil.

Mit der Type-Traits Bibliothek und den konstanten Ausdrücken verliert Programmierung zur Compilezeit seinen Charakter als Werkzeug für den C++-Experten. Programmierung zur Compilezeit wird damit zu einem mächtigen Werkzeug in der Werkzeugbox des C++-Entwicklers, um korrekte und performante Programme zu schreiben.

Weiterführende Information

[1]: Turing-Vollständigkeit: https://de.wikipedia.org/wiki/Turing-Vollst%C3%A4ndigkeit
[2]: Boost-Bibliothek: http://www.boost.org/

Rainer Grimm, „C++11 für Programmierer“, O’Reilly 2013
Rainer Grimm, „Modernes C++ in der Praxis“, Serie im Linux-Magazin seit 2011

Autor

Rainer Grimm ist seit vielen Jahren als Softwareentwickler und Schulungsleiter tätig. In seiner Freizeit schreibt er gerne Artikel zu den Programmiersprachen C++, Python und Haskell. Seit 2013 ist er als Softwarearchitekt und Gruppenleiter bei der schwäbischen Firma Metrax beschäftigt. Insbesondere die Software auf den Defibrillatoren ist ihm eine Herzensangelegenheit. So bringt er nach 30 Jahren seine Ausbildungen als Krankenpfleger mit der eines Mathematikers wieder zusammen. Seine Bücher "C++11 für Programmierer", "C++" und "C++-Standardbibliothek" für die kurz und gut Reihe sind beim Verlag O'Reilly erschienen.