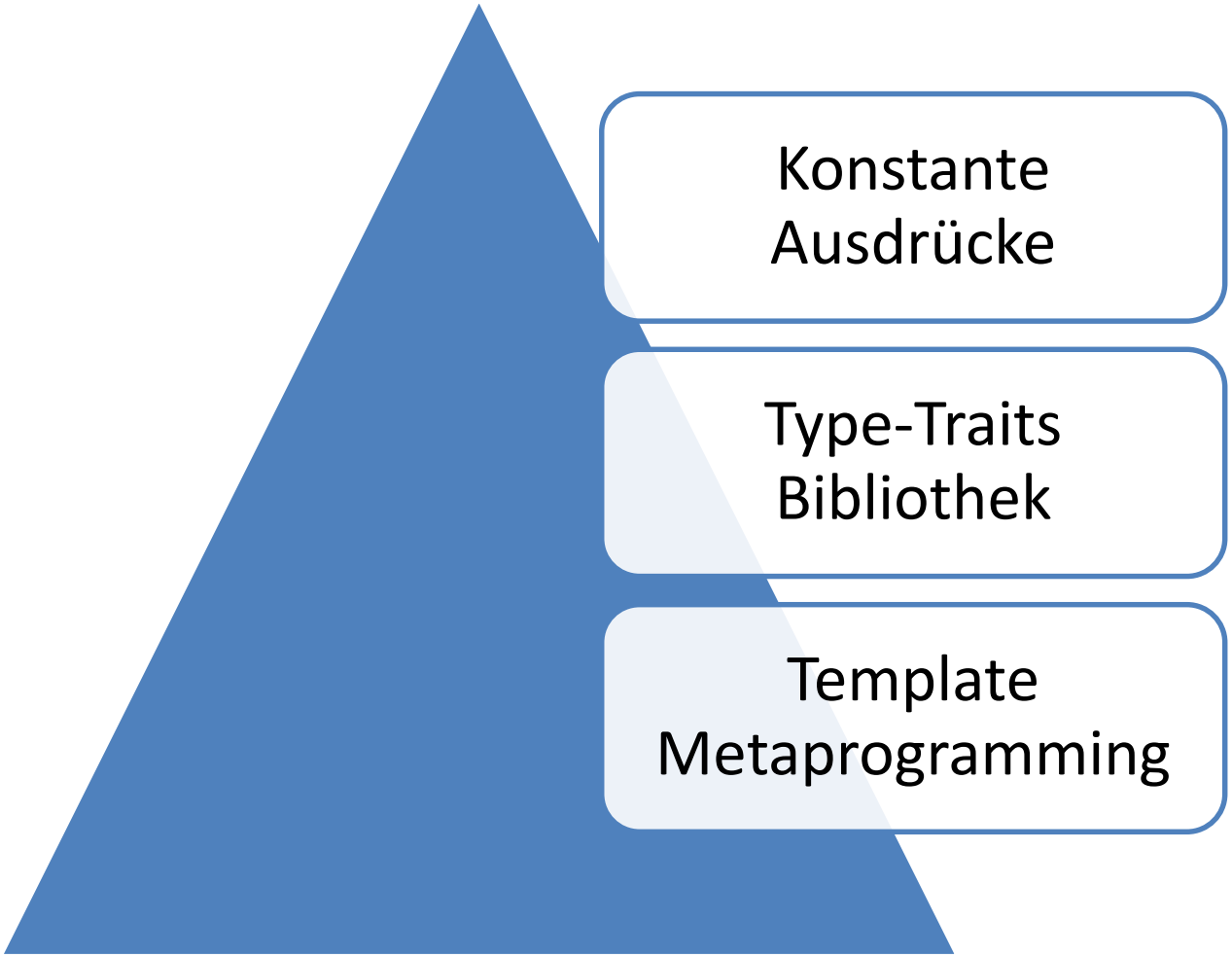
Advanced Developers Conference C++
Development for Professionals!

Programmierung zur Compilezeit

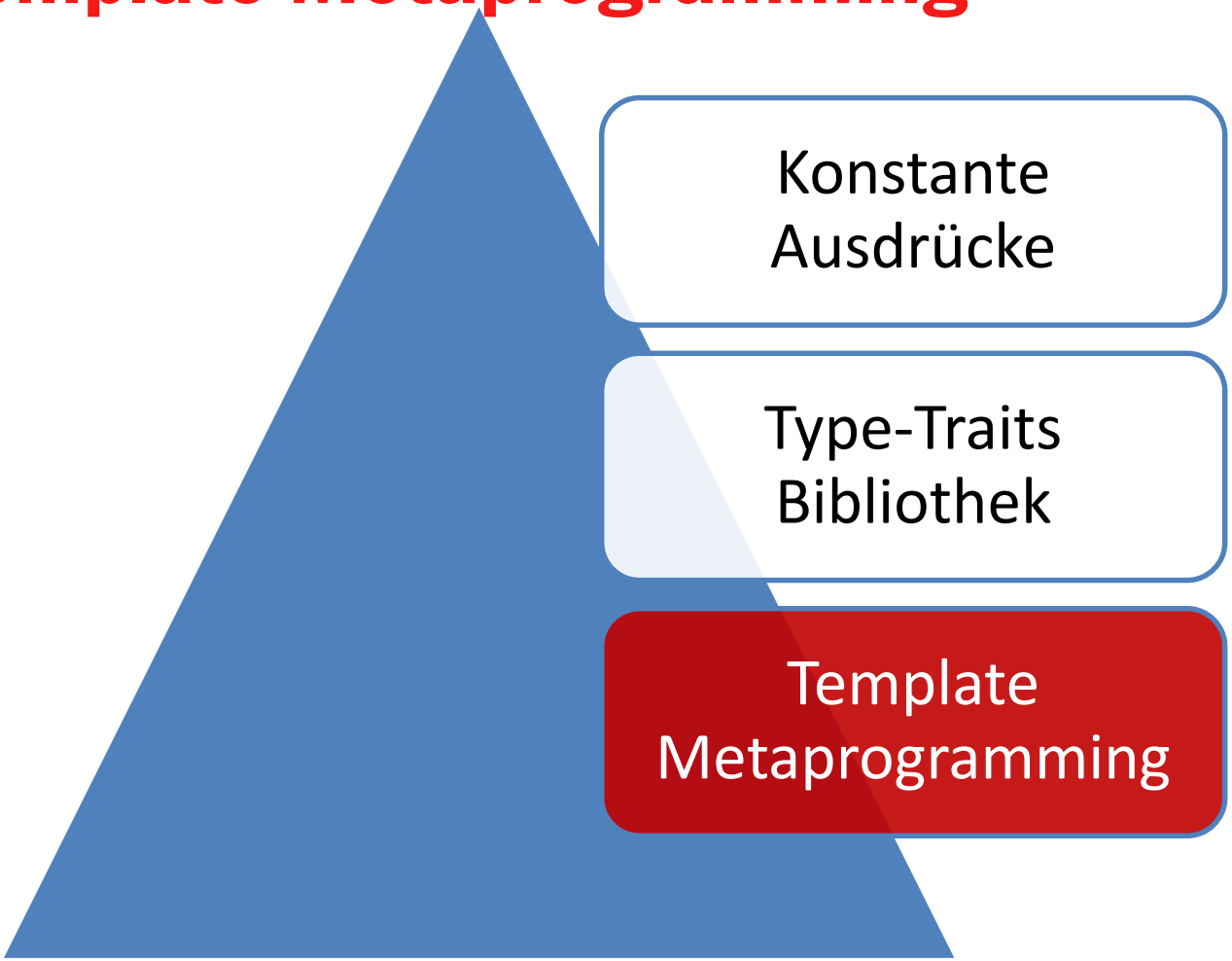
Rainer Grimm

Veranstalter
ppedv
Gold-Partner
Microsoft

Überblick


Konstante Ausdrücke
Type-Traits Bibliothek
Template Metaprogramming

Template Metaprogramming



Wie alles begann

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

Die Fehlerausgabe

```
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).
```

Metaprogramming

Compilezeit

```
int main(){
  cout << Factorial<4>::value;
  cout << Factorial<5>::value;
}
```

Template Instanziierung

```
int main(){
  cout << 24;
  cout << 120;
}
```

Laufzeit

```
mov $0x18,%esi
mov $0x601060,%edi
...
mov $0x78,%esi
mov $0x601060,%edi
...
```

Ausführung

Der Klassiker

```
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;

Factorial<5>::value
   → 5*Factorial<4>::value
   → 5*4*Factorial<3>::value
   → 5*4*3*Factorial<2>::value
   → 5*4*3*2*Factorial<1>::value → 5*4*3*2*1= 120
```

Die Magie

Datei Bearbeiten Ansicht Verlauf Lesezeichen Einstellungen Hilfe

```
int main(){
  400874:        55                          push   %rbp
  400875:        48 89 e5                    mov    %rsp,%rbp
    std::cout << Factorial<5>::value << std::endl;
  400878:        be 78 00 00 00              mov    $0x78,%esi
  40087d:        bf 60 10 60 00              mov    $0x601060,%edi
  400882:        e8 89 fe ff ff              callq  400710 <_ZNSolsEi@plt>
  400887:        be 70 07 40 00              mov    $0x400770,%esi
  40088c:        48 89 c7                    mov    %rax,%rdi
  40088f:        e8 cc fe ff ff              callq  400760 <_ZNSolsEPFRSoS_E@plt>
    std::cout << 120 << std::endl;
  400894:        be 78 00 00 00              mov    $0x78,%esi
  400899:        bf 60 10 60 00              mov    $0x601060,%edi
  40089e:        e8 6d fe ff ff              callq  400710 <_ZNSolsEi@plt>
  4008a3:        be 70 07 40 00              mov    $0x400770,%esi
  4008a8:        48 89 c7                    mov    %rax,%rdi
  4008ab:        e8 b0 fe ff ff              callq  400760 <_ZNSolsEPFRSoS_E@plt>
    return 0;
  4008b0:        b8 00 00 00 00              mov    $0x0,%eax
}
```

source : bash

Arbeiten mit Typen

```
template <typename T>
struct RemoveConst{
  typedef T type;
};

template <typename T>
struct RemoveConst<const T>{
  typedef T type;
};

int main(){
  std::is_same<int,RemoveCost<int>::type>::value;        // true
  std::is_same<int,RemoveConst<const int>::type>::value  // true
}
```

Metafunktionen

- **Metadaten**: Typen und integrale Daten, die von den Metafunktionen verwendet werden.
- **Metafunktionen**: Funktionen, die zur Compilezeit ausgeführt wird.
  - Geben ihren Wert per `::value` zurück.

```
template <>
struct Factorial<1>{
  static int const value = 1;
};
```

  - Geben ihren Typ per `::type` zurück.

```
template <typename T>
struct RemoveConst<const T>{
  typedef T type;
};
```

Metafunktionen

Charakteristiken	Funktionen	Metafunktionen
Beispiel	`power(2,10)`	`Power<2,10>::value`
Ausführungszeit	Runtime	Compilezeit
Argumente	Funktionsargumente	Templateargumente
Argumente und Rückgabewerte	Beliebige Daten	Typen, Nicht-Typen(integrale Daten) und Klassen-Templates
Implementierung	Aufrufbare Einheiten	Klassen-Templates
Daten	Veränderlich	Nicht veränderlich
Veränderung	Daten werden modifiziert	Neue Daten werden erzeugt

Metafunktionen

Funktion

```
int power(int m, int n){
  int r = 1;
  for(int k=1; k<=n; ++k) r*= m;
  return r;
}
```

Metafunktion

```
template<int m, int n>
struct Power{
  static int const value = m * Power<m,n-1>::value;
};

template<int m>
struct Power<m,0>{
  static int const value = 1;
};
```

```
int main(){
  std::cout << power(2,10) << std::endl;          // 1024
  std::cout << Power<2,10>::value << std::endl;   // 1024
}
```

Metafunktion oder Funktion?

```
template<int n>
int power(int m){
  return m * power<n-1>(m);
}
template<>
int power<1>(int m){
  return m;
}
template<>
int power<0>(int m){
  return 1;
}
int main(){
  std::cout << power<10>(2) << std::endl;          // 1024
}
```

power<10>(2) → m*m*m*m*m*m*m*m*m*m → 2*2*2*2*2*2*2*2*2*2= 1024

Rein funktionale Subsprache

- Template Metaprogramming ist
  - eine eingebettete, rein funktionale Subsprache in der imperativen Sprache C++.
  - Turing-vollständig.
  - eine intellektuelle Spielwiese für C++-Experten.
  - die Grundlage für viele Boost-Bibliotheken.

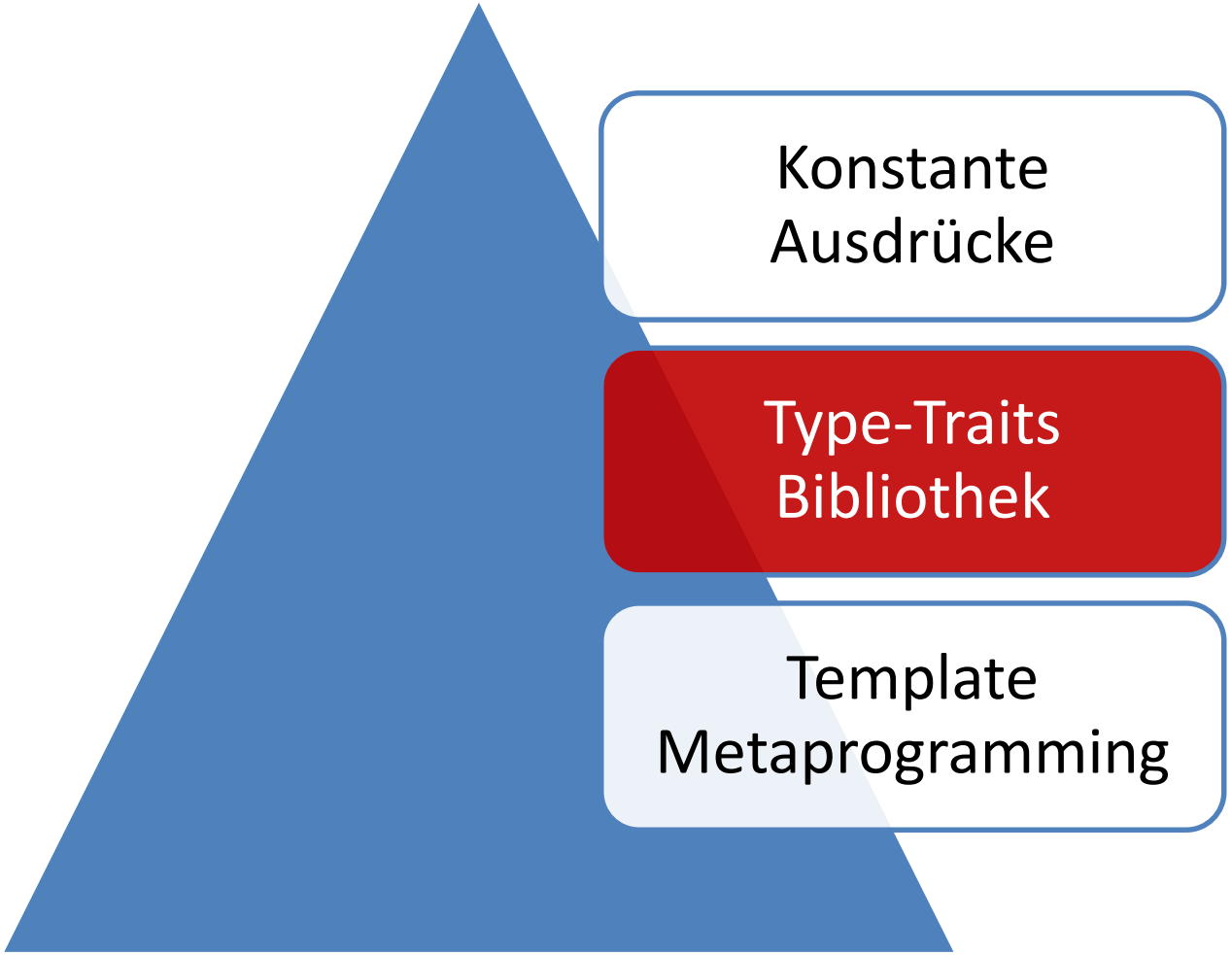
Die Template-Rekursions-Tiefe ist beschränkt.

- C++03: 17
- C++11: 1024

Weitere Informationen

- Bücher
  - Generative Programming: Methods, Tools and Applikation von Krysztof Czarnecki und Ulrich Eisenecker
  - Modern C++-Design von Andrei Alexandrescu
  - Advanced C++ Metaprogramming von Davide Di Gennaro
- The Boost MPL Library von Aleksey Gurtovoy und David Abrahams
- Template Metaprogramming mit C++ von Rainer Grimm

Type-Traits Bibliothek



Type-Traits Bibliothek

- Ermöglicht Typabfragen, Typvergleiche und Typtransformationen.
- Anwendung von Template Metaprogramming
  - Programmierung zur Compilezeit
  - Programmierung auf Typen und nicht auf Werten
  - Compiler interpretiert die Templates und transformiert diese in C++-Quelltext
- Ziele
  - Korrektheit und Optimierung

Type-Traits Bibliothek

- Typabfragen (`::value`)
  - Primäre Typkategorien

    `std::is_pointer<T>, std::is_integral<T>`

    `std::is_class<T>, std::is_floating_point<T>`
  - Zusammengesetzte Typkategorien

    `std::is_arithmetic<T>, std::is_object<T>`
  - Typeigenschaften

    `std::is_const<T>, std::is_abstract<T>,`

    `std::is_pod<T>, std::is_signed<T>`
- Typvergleiche (`::value`)

  `std::is_same<T,U>, std::is_base_of<Base,Derived>`

  `std::is_convertible<From,To>`

Type-Traits Bibliothek

- Typtransformationen (`::type`)

  `std::add_const<T>, std::remove_reference<T>`

  `std::make_signed<T>, std::make_unsigned<T>`

  - Verschiedene (`::type`)

    `std::enable_if<bool,T>`
    `std::conditional<bool,T,F>`

    `std::common_type<T1,T2,T3, ... >`

- Weitere Informationen:
  - type_traits

Type-Traits Bibliothek

```
template <typename T>
struct add_const{
  typedef const T type;
};

template <typename T>
struct remove_const {
  typedef T type;
};

template <typename T >
struct remove_const <const T> {
  typedef T type;
};
```

```
template<class T, class U>
struct is_same: std::false_type{};

template<class T>
struct is_same<T, T>: std::true_type{};
```

```
std::cout << is_same<add_const<int>::type,const int>::value ;    // true
std::cout << is_same<remove_const<const int>::type,int>::value;  // true
```

Type-Traits: Korrektheit

- Typinformationen werden zur Compilezeit ausgewertet.
- Die ausgewerteten Typinformationen definieren mit `static_assert` verbindliche Zusicherungen an den Code.

Type-Traits: Korrektheit

```
#include <iostream>

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
}
```

Type-Traits: Korrektheit

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

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

int main(){
  std::cout << gcd(3.5,4.0) << std::endl;
  std::cout << gcd("100","10") << std::endl;
}
```

rainer : bash - Konsole <2>

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@icho:~> g++ -std=c++11 -O3 -o saveGcd saveGcd.cpp
saveGcd.cpp: In instantiation of 'T gcd(T, T) [with T = double]':
saveGcd.cpp:18:27:   required from here
saveGcd.cpp:7:3: error: static assertion failed: T should be an integral type!
saveGcd.cpp:10:24: error: invalid operands of types 'double' and 'double' to binary 'operator%'
saveGcd.cpp: In instantiation of 'T gcd(T, T) [with T = const char*]':
saveGcd.cpp:19:30:   required from here
saveGcd.cpp:7:3: error: static assertion failed: T should be an integral type!
saveGcd.cpp:10:24: error: invalid operands of types 'const char*' and 'const char*' to binary '
rainer@icho:~>
```

rainer : bash

Type-Traits: Korrektheit

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

template<typename T1, typename T2>
??? 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);
}

int main(){
  std::cout << gcd(100,10L) << std::endl;
}
```

Type-Traits: Korrektheit

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

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);
}

int main(){
  std::cout << gcd(100,10LL) << std::endl;
  auto res= gcd(100,10LL);
  auto res2= gcd(100LL,10L);
  std::cout << typeid(res).name() << std::endl;    // i
  std::cout << typeid(res2).name() << std::endl;   // l
  std::cout << std::endl;
}
```

Type-Traits: Korrektheit

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

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

int main(){

  std::cout << typeid(gcd(100,10)).name() << std::endl;            // i
  std::cout << typeid(gcd(100,10L)).name() << std::endl;           // l
  std::cout << typeid(gcd(100,10LL)).name() << std::endl;          // x

}
```

Type-Traits: Korrektheit

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

template<typename T1,
         typename T2,
         typename std::enable_if<std::is_integral<T1>::value,T1 >::type= 0,
         typename std::enable_if<std::is_integral<T2>::value,T2 >::type= 0,
     typename R = typename std::conditional<(sizeof(T1) < sizeof(T2)),T1,T2>::type>
R gcd(T1 a, T2 b){
  if( b == 0 ){ return a; }
  else{
    return gcd(b, a % b);
  }
}

int main(){

  std::cout << "gcd(100,10)= " << gcd(100,10)  << std::endl;
  std::cout << "gcd(100,33)= " << gcd(100,33) << std::endl;
  std::cout << "gcd(3.5,4.0)= " << gcd(3.5,4.0) << std::endl;

}
```

Type-Traits: Korrektheit

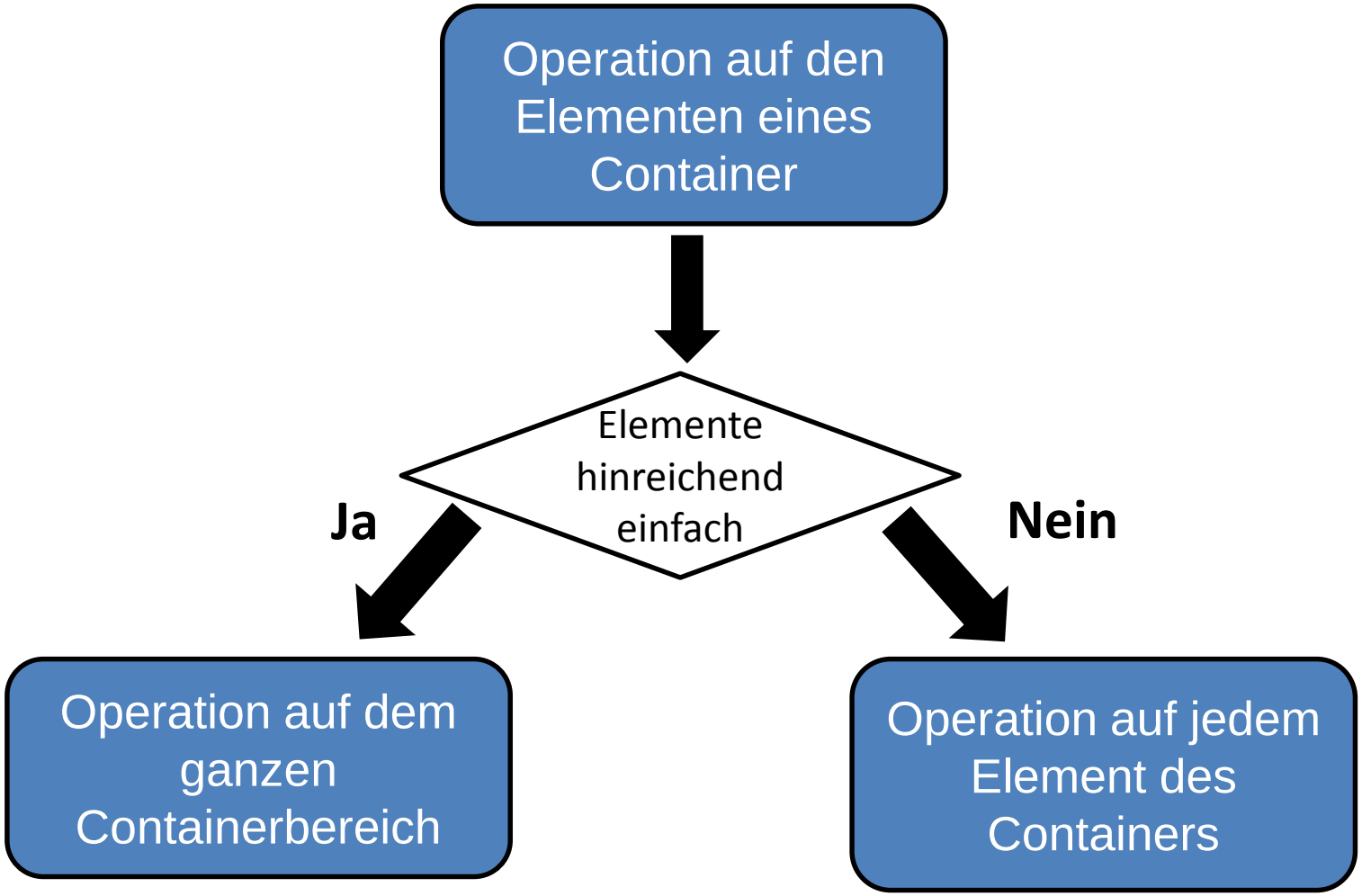
```
std::cout << "gcd(3.5,4.0)= " << gcd(3.5,4.0) << std::endl;
```

```
src : bash - Konsole
File  Edit  View  Bookmarks  Settings  Help
rainer@linux:~>g++ -std=c++11 -o gcd gcd.cpp
gcd.cpp: In function 'int main()':
gcd.cpp:22:45: error: no matching function for call to 'gcd(double, double)'
   std::cout << "gcd(3.5,4)= " << gcd(3.5,4.0) << std::endl;   // ERROR
                                             ^
gcd.cpp:22:45: note: candidate is:
gcd.cpp:9:3: note: template<class T1, class T2, typename std::enable_if<std::is_integral<_Tp>::value, T1>::type <anonymous>,
 typename std::enable_if<std::is_integral<T2>::value, T2>::type <anonymous>, class R> R gcd(T1, T2)
 R gcd(T1 a, T2 b){
   ^
gcd.cpp:9:3: note:   template argument deduction/substitution failed:
gcd.cpp:6:74: error: no type named 'type' in 'struct std::enable_if<false, double>'
          typename std::enable_if<std::is_integral<T1>::value,T1 >::type= 0,
                                                                        ^
gcd.cpp:6:74: note: invalid template non-type parameter
rainer@linux:~>
src : bash
```

Type-Traits: Optimierung

- Code, der sich beim Übersetzen selbst optimiert.
  - ➡ Abhängig vom Typ einer Variable wird ein besonderer Algorithmus ausgewählt.
- Die STL enthält optimierte Versionen von `std::copy`, `std::fill` oder `std::equal`.
  - ➡ Algorithmen können direkt auf den Speicherbereichen angewandt werden.

Type-Traits: Optimierung

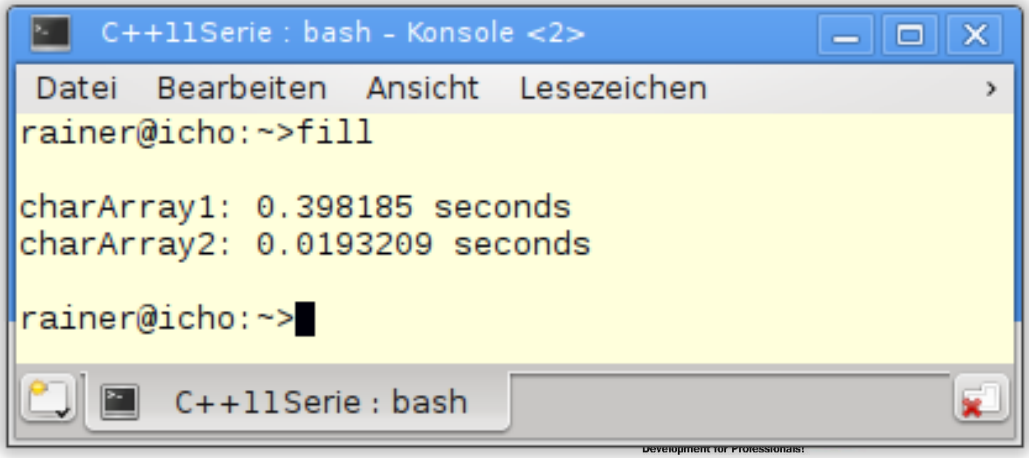


Type-Traits: `std::fill`

```
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 std::integral_constant<bool,std::is_trivially_copy_assignable<T>::value
                                 && (sizeof(T) == 1)> boolType;
  fill_impl(first, last, val, boolType());
}
```

Type-Traits: `std::fill`

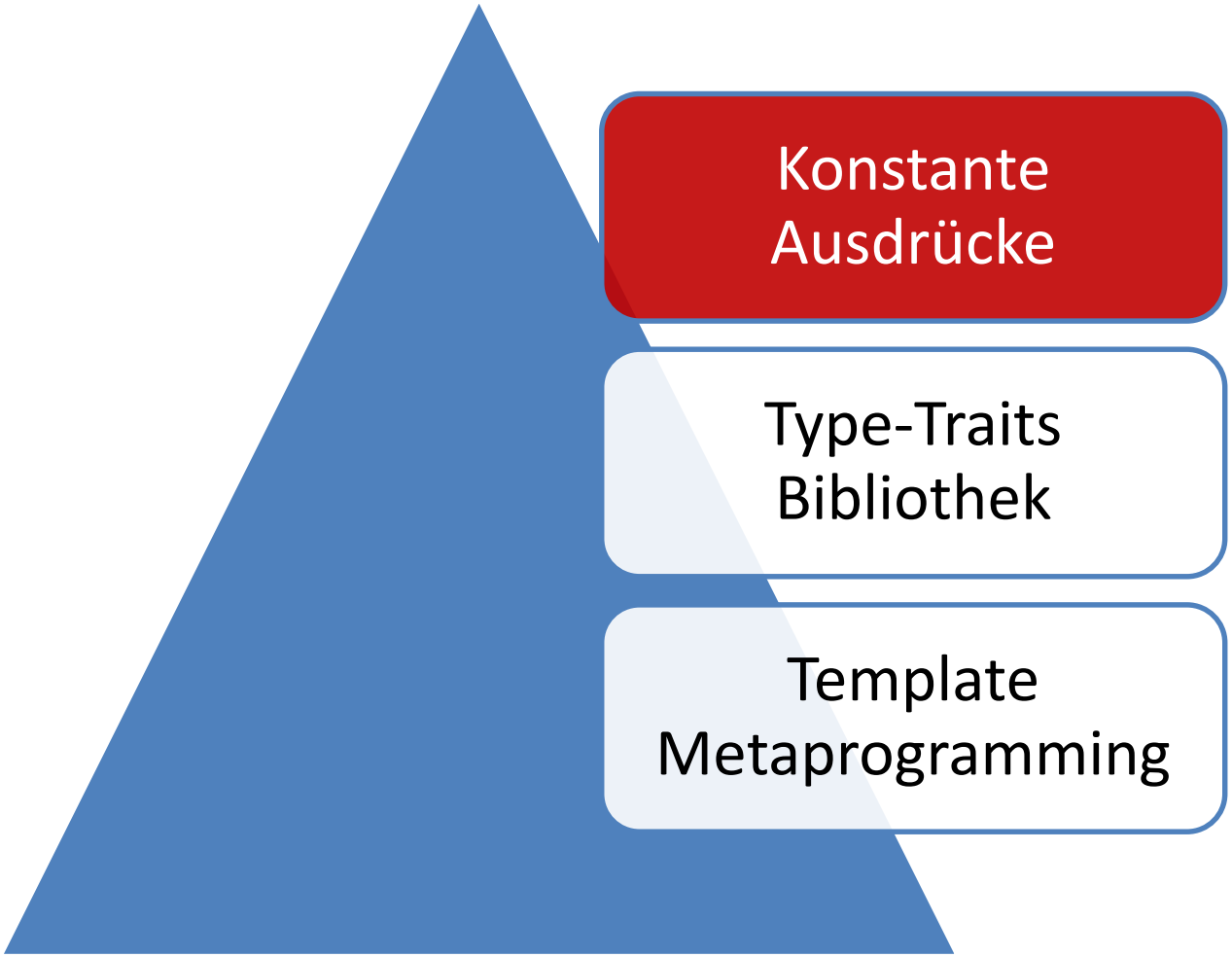
```
const int arraySize = 100'000'000;
char charArray1[arraySize]= {0,};
char charArray2[arraySize]= {0,};

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



Type-Traits: `std::equal`

```
template<typename _II1, typename _II2>
inline bool __equal_aux(_II1 __first1, _II1 __last1, _II2 __first2){
  typedef typename iterator_traits<_II1>::value_type _ValueType1;
  typedef typename iterator_traits<_II2>::value_type _ValueType2;
  const bool __simple = ((__is_integer<_ValueType1>::__value
                          || __is_pointer<_ValueType1>::__value )
                         && __is_pointer<_II1>::__value
                         && __is_pointer<_II2>::__value
                         &&__are_same<_ValueType1, _ValueType2>::__value
                        );
  return std::__equal<__simple>::equal(__first1, __last1, __first2);
 }
```

Konstante Ausdrücke



Konstante Ausdrücke

- Konstante Ausdrücke
  - können zur Compilezeit evaluiert und im ROM gespeichert werden.
  - geben dem Compiler tiefen Einblick in den Code.
  - sind implizit threadsicher.
  - gibt es als Variablen, benutzerdefinierte Typen und Funktionen.

➡ Erlaubt das Programmieren zur Compilezeit im imperativen Stil.

Konstante Ausdrücke

1. Variablen
   - sind implizit const.
   - müssen durch einen konstanten Ausdruck initialisiert werden.

```
constexpr double pi= 3.14;
```

2. Benutzerdefinierte Typen
   - Können keine virtuelle Basisklasse besitzen.
   - Der Konstruktor
     - muss leer sein.
     - muss ein konstanter Ausdruck sein.
   - Die Methoden
     - muss ein konstanter Ausdruck sein.
     - können nicht `virtual` sein und sind implizit `const` (C++11).

➡ Können zur Compilezeit initialisiert werden.

Konstante Ausdrücke

3. Funktionen (C++11)
   - können nur einen Wert zurückliefern.
   - werden zur Compilezeit ausgewertet, wenn sie mit einem konstanten Ausdruck aufgerufen werden.
   - können nur einen Funktionskörper besitzen, der aus einer Rückgabeanweisung bestehen und diese muss selbst ein konstanter Ausdruck sein.
   - sind implizit `inline`.

➡ Sehr eingeschränkte Funktionen, die nur einen konstanten Ausdruck zurückgeben.

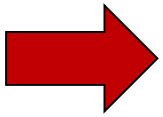
Konstante Ausdrücke

3. Funktionen (C++14)
   - können bedingte Sprung- und Iterationsanweisungen enthalten.
   - können Variablen enthalten, die mit einem konstanten Ausdruck initialisiert werden müssen.
   - können keine statische oder `thread_local` Variablen enthalten.

➡ Normale Funktionen mit ein paar Einschränkungen.

Konstante Ausdrücke

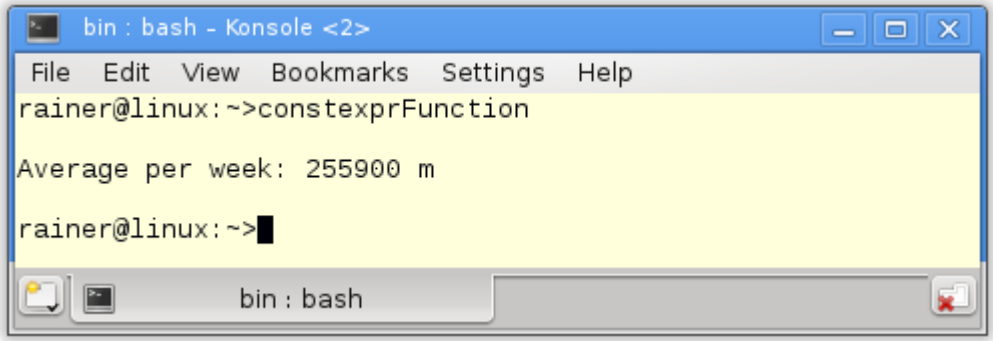
```
#include <iostream>

constexpr int gcd11(int a, int b){
  return ( b == 0) ? a : gcd11(b, a % b);
}
constexpr auto gcd14(int a, int b){
  while (b != 0){
    auto t= b;
    b= a % b;
    a= t;
  }
  return a;
}

int main(){
  constexpr int i= gcd11(11,121);
  std::cout << i << std::endl;            // 11
  constexpr int j= gcd14(100,1000);
  std::cout << j << std::endl;            // 100
}
```



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

Konstante Ausdrücke

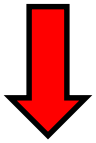
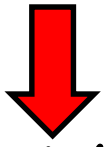
```
int main(){

  constexpr Dist work= 63.0_km;
  constexpr Dist workPerDay= 2 * work;
  constexpr Dist abbreToWork= 5400.0_m;          // abbrevation to work
  constexpr Dist workout= 2 * 1600.0_m;
  constexpr Dist shop= 2 * 1200.0_m;             // shopping

  constexpr Dist distPerWeek1= 4*workPerDay - 3*abbreToWork + workout + shop;
  constexpr Dist distPerWeek2= 4*workPerDay - 3*abbreToWork + 2*workout;
  constexpr Dist distPerWeek3= 4*workout + 2*shop;
  constexpr Dist distPerWeek4= 5*workout + shop;

  constexpr Dist perMonth=  getAverageDistance({distPerWeek1,
                            distPerWeek2,distPerWeek3,distPerWeek4});

  std::cout << "Average per week: " << averagePerWeek << std::endl;
}
```



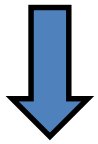
Konstante Ausdrücke

```
1.5_km      +    105.1_m
```

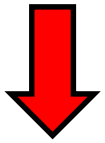

```
operator"" _km(1.5)    +    operator"" _m(105.1)
```

```
Dist(1500.0)   +    Dist(105.1)
```

```
operator+ (1500.0,105.1)
```



```
Dist(1605.1)
```

Konstante Ausdrücke

```
namespace Unit{
  Dist constexpr operator "" _km(long double d){
    return Dist(1000*d);
  }
  Dist constexpr operator "" _m(long double m){
    return Dist(m);
  }
  Dist constexpr operator "" _dm(long double d){
    return Dist(d/10);
  }
  Dist constexpr operator "" _cm(long double c){
    return Dist(c/100);
  }
}
constexpr Dist getAverageDistance(std::initializer_list<Dist> inList){
  auto sum= Dist(0.0);
  for ( auto i: inList) sum += i;
  return sum/inList.size();
}
```

Konstante Ausdrücke

```
class Dist{
public:
  constexpr Dist(double i):m(i){}

  friend constexpr Dist operator +(const Dist& a, const Dist& b){
    return Dist(a.m + b.m);
  }
  friend constexpr Dist operator -(const Dist& a,const Dist& b){
    return Dist(a.m - b.m);
  }
  friend constexpr Dist operator*(double m, const Dist& a){
    return Dist(m*a.m);
  }
  friend constexpr Dist operator/(const Dist& a, int n){
    return Dist(a.m/n);
  }
  friend std::ostream& operator<< (std::ostream &out, const Dist& myDist){
    out << myDist.m << " m";
    return out;
  }
private:
  double m;
};
```

Konstante Ausdrücke

Charakteristiken	Konstante Ausdrücke	Template Metaprogramming
Ausführungszeit	Compilezeit/Laufzeit	Compilezeit
Argumente	Werte	Typen und Werte
Programmierparadigma	Imperativ	Funktional
Veränderung	Ja	Nein
Kontrollstruktur	Sprung- und Iterationsanweisungen	Rekursion
Bedingte Ausführung	Bedingte Anweisungen	Template-Spezialisierung

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

???

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

Konstante Ausdrücke

```
constexpr int factorial(int n){
  auto res= 1;
  for ( auto i= n; i >= 1; --i ){
    res *= i;
  }
  return res;
}
```

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

template <>
struct Factorial<1>{
  static int const value = 1;
};
```

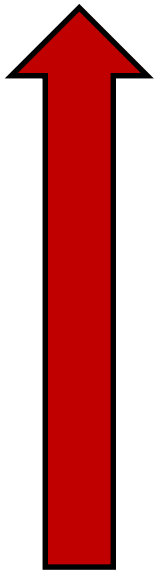
Konstante Ausdrücke


```
constexpr double avera(double fir, double sec){
   return (fir+sec)/2;
}

int main(){
  constexpr double res= avera(2,3);
}
```

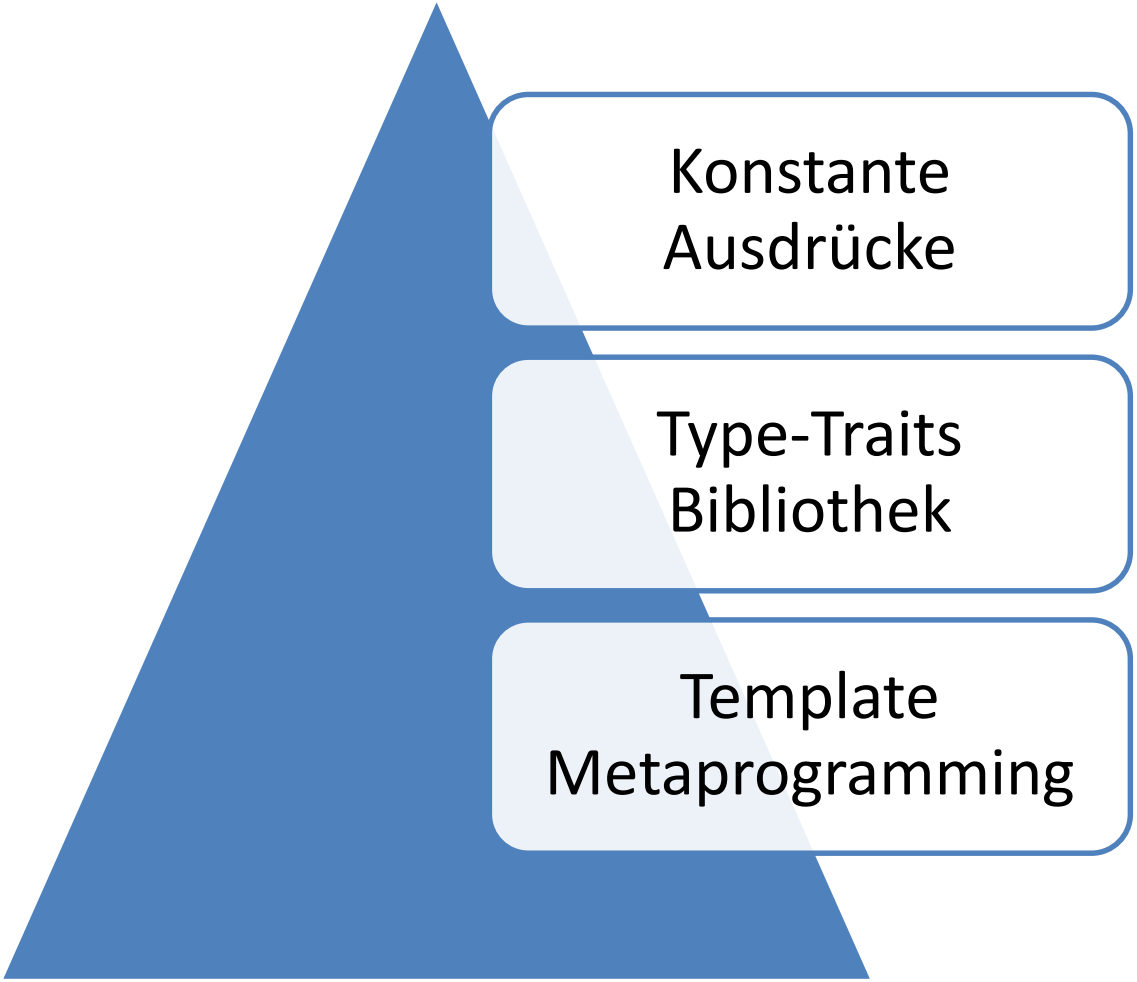
```
pushq    %rbp
movq     %rsp, %rbp
movabsq  $4612811918334230528, %rax
movq     %rax, -8(%rbp)
movl     $0, %eax
popq     %rbp
```

Fazit

Imperativ



Funktional



Vielen Dank!

Rainer Grimm