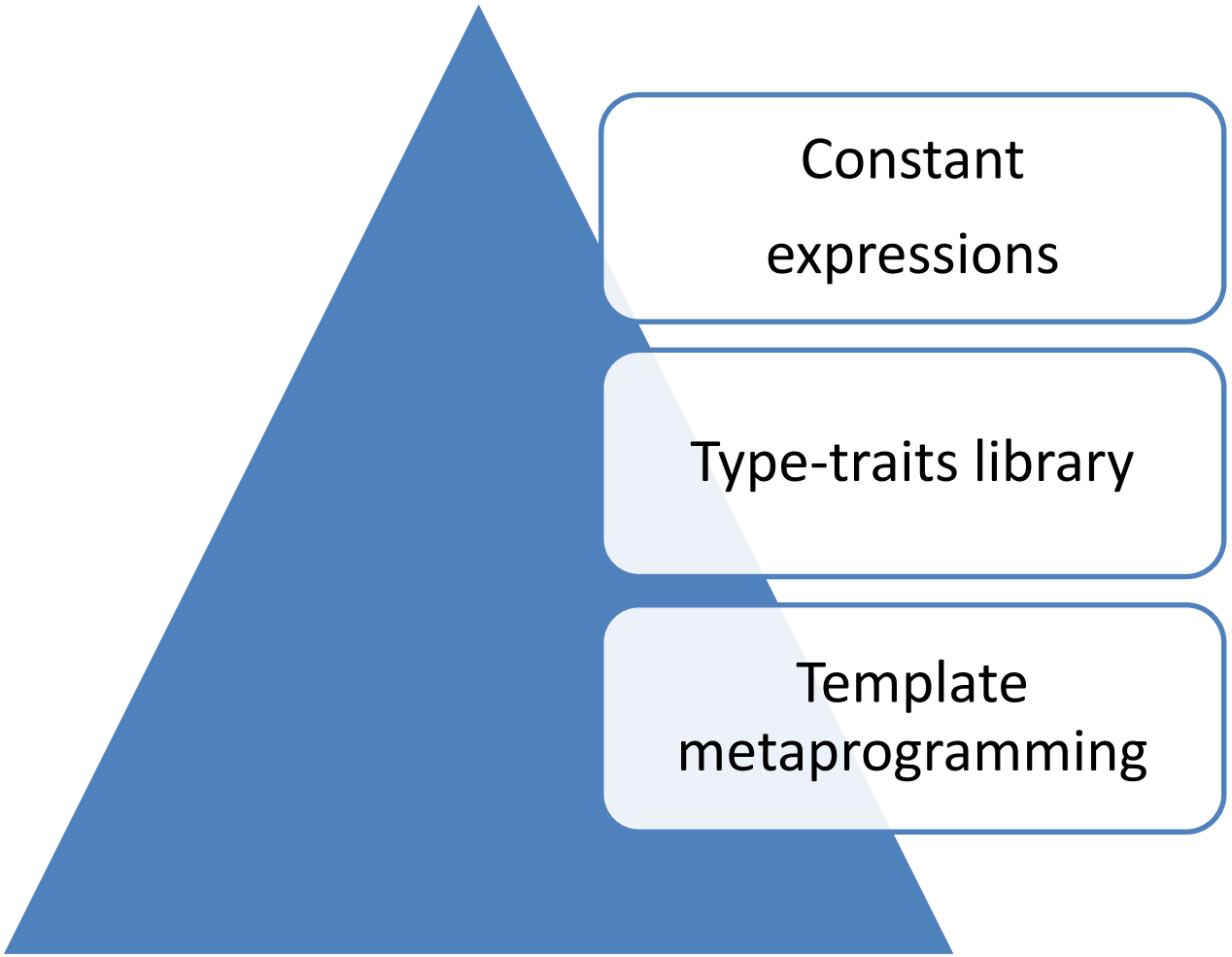
Programming at Compile Time

Rainer Grimm

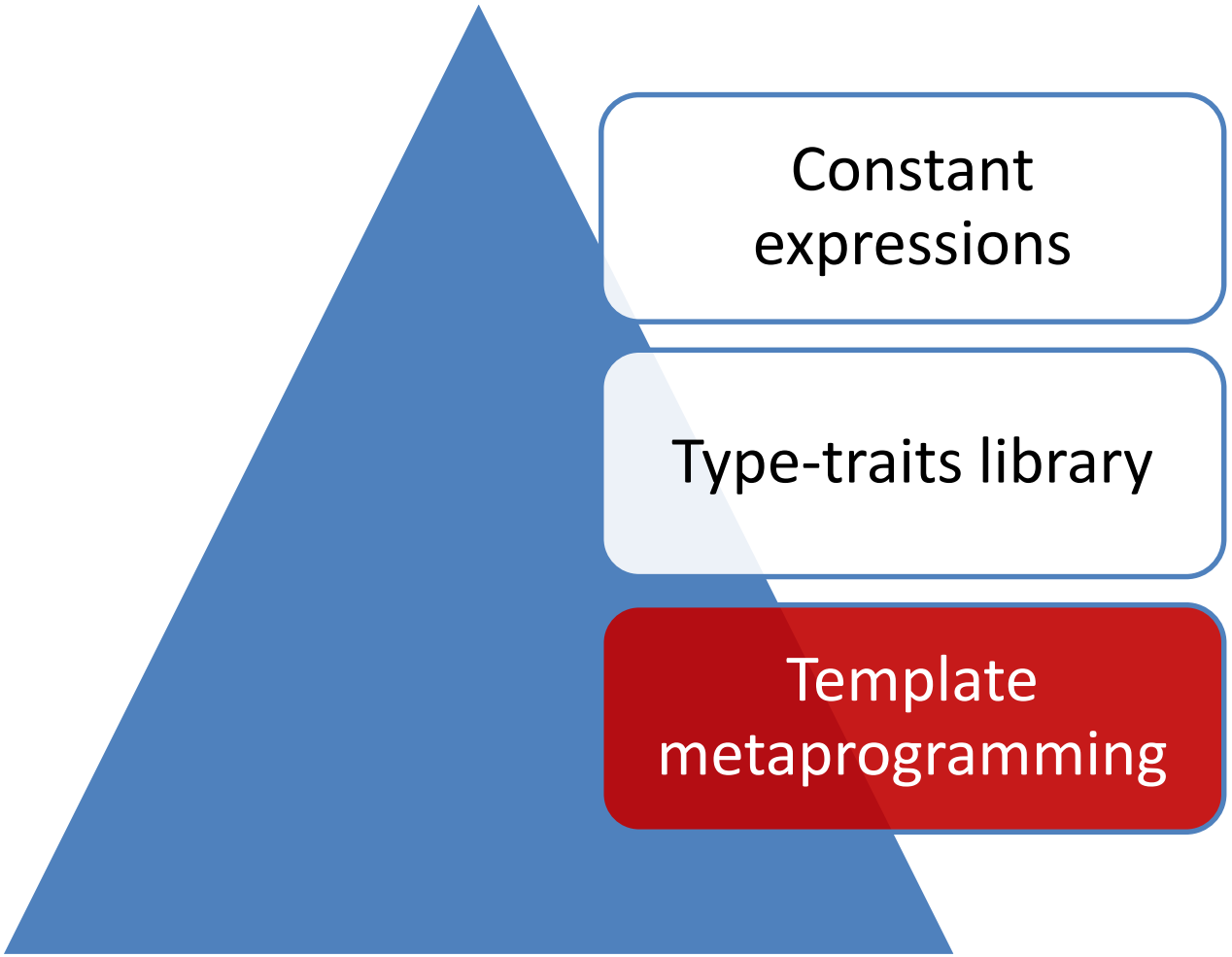
Training, Coaching, and Technology Consulting

www.ModernesCpp.de

Overview


Constant expressions
Type-traits library
Template metaprogramming

Template Metaprogramming



A long time ago

- 1994 Erwin Unruh discovered by accident template metaprogramming.
- His program calculated at compile time the first 30 prime numbers.
- The output of the prime numbers were part of the error message.

The Error Message

```
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

Compile time

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

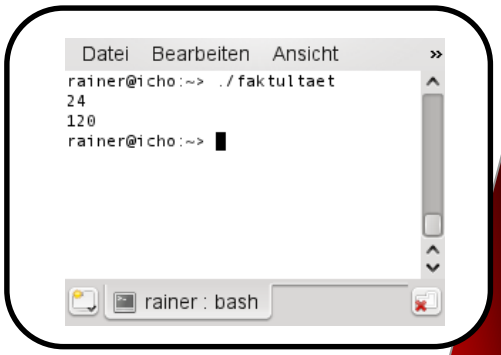
Template Instantiation

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

Run time

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

Execution



The Classic

```
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

The Magic

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
int main(){
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp

  std::cout << Factorial<5>::value << std::endl;
   4:   be 78 00 00 00          mov    $0x78,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   be 00 00 00 00          mov    $0x0,%esi
  18:   48 89 c7                mov    %rax,%rdi
  1b:   e8 00 00 00 00          callq  20 <main+0x20>
  std::cout << 120 << std::endl;
  20:   be 78 00 00 00          mov    $0x78,%esi
  25:   bf 00 00 00 00          mov    $0x0,%edi
  2a:   e8 00 00 00 00          callq  2f <main+0x2f>
  2f:   be 00 00 00 00          mov    $0x0,%esi
  34:   48 89 c7                mov    %rax,%rdi
  37:   e8 00 00 00 00          callq  3c <main+0x3c>

}
```

Austausch : bash

Meta Functions

- **Meta data**: Types and integral types that are used in meta functions.
- **Meta functions**: Functions that are executed at compile time.
  - Return their value by `::value`.

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

  - Return their type by `::type`.

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

Meta Functions

Function

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

Meta function

```
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
}
```

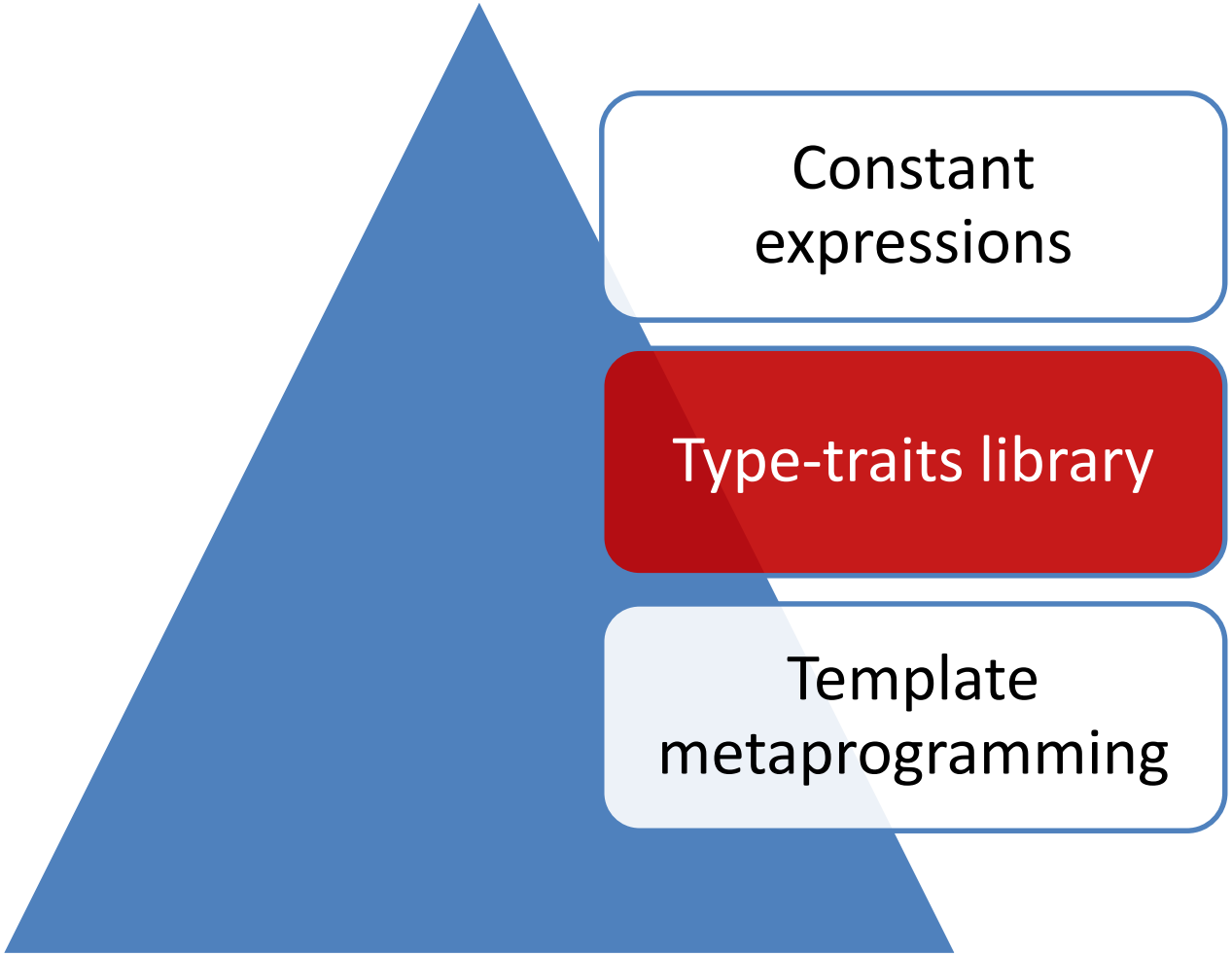
Pure Functional Sublanguage

- Template metaprogramming is
  - an embedded pure functional language in the imperative language C++.
  - Turing-complete.
  - an intellectual playground for C++ experts.
  - the foundation for many boost libraries.

The template recursion depth is limited.

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

Type-Traits Library



Type-Traits Library

- Enables type checks, type comparisons, and type modifications at compile time
- Application of template metaprogramming
  - Programming at compile time
  - Programming with types and values
  - Compiler translates the templates and transforms it in C++ source code
- Goals
  - Correctness and optimization

Type-Traits Library

- Type checks
  - Primary type category (`::value`)

    `std::is_pointer<T>, std::is_integral<T>, std::is_floating_point<T>`
  - Composed type category (`::value`)

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

- Type comparisons (`::value`)

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

Type-Traits Library

- Type transformation (`::type`)

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

  - Others (`::type`)

```
std::enable_if<bool,T> std::conditional<bool,T,F>
std::common_type<T1,T2,T3, ... >
```

- Further information
  - type_traits

Type-Traits: Correctness

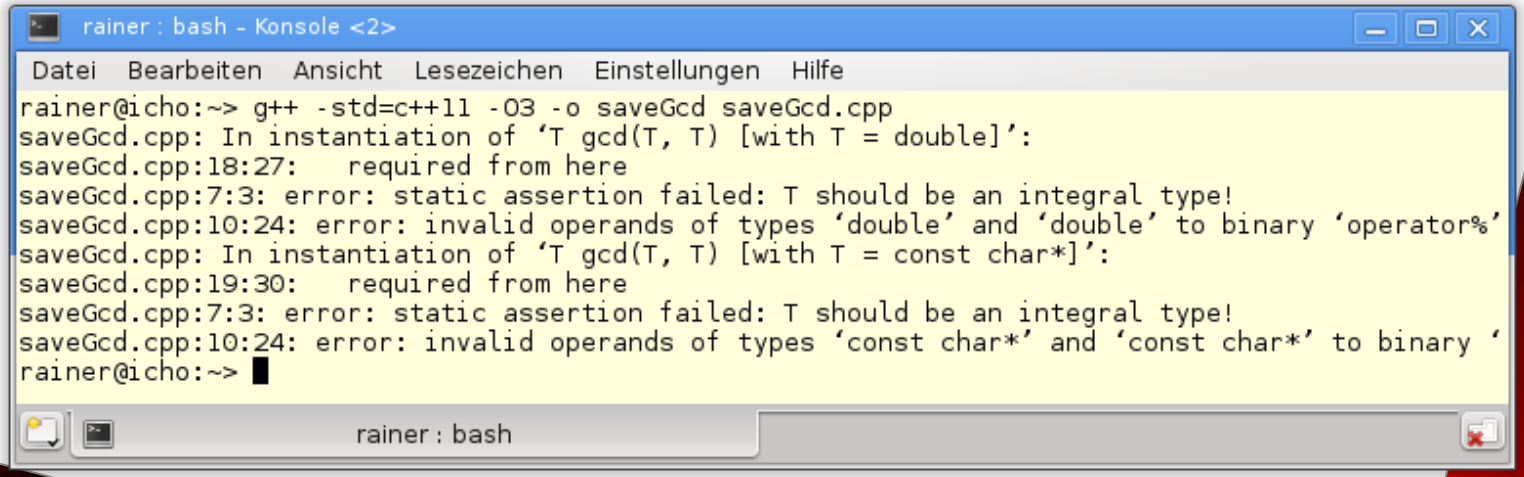
- Type information is evaluated a compile time.
- The evaluated type information can be checked with `static_assert`.

Type-Traits: Correctness

```
#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: Correctness

```
#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;
}
```



Type-Traits: Correctness

```
#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: Correctness

```
#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);
  std::conditional<(sizeof(long long)<sizeof(long)), long long, long>::type 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: Correctness

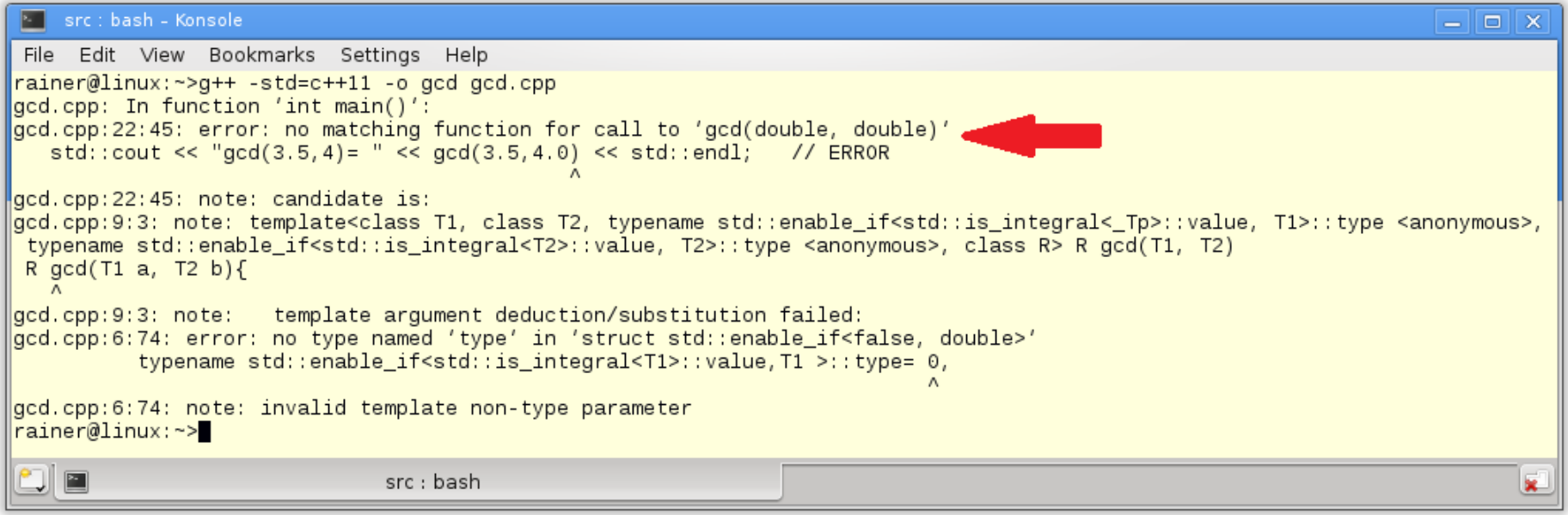
```
#include <iostream>
#include <type_traits>
#include <typeinfo>

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: Correctness

```
#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: Correctness

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

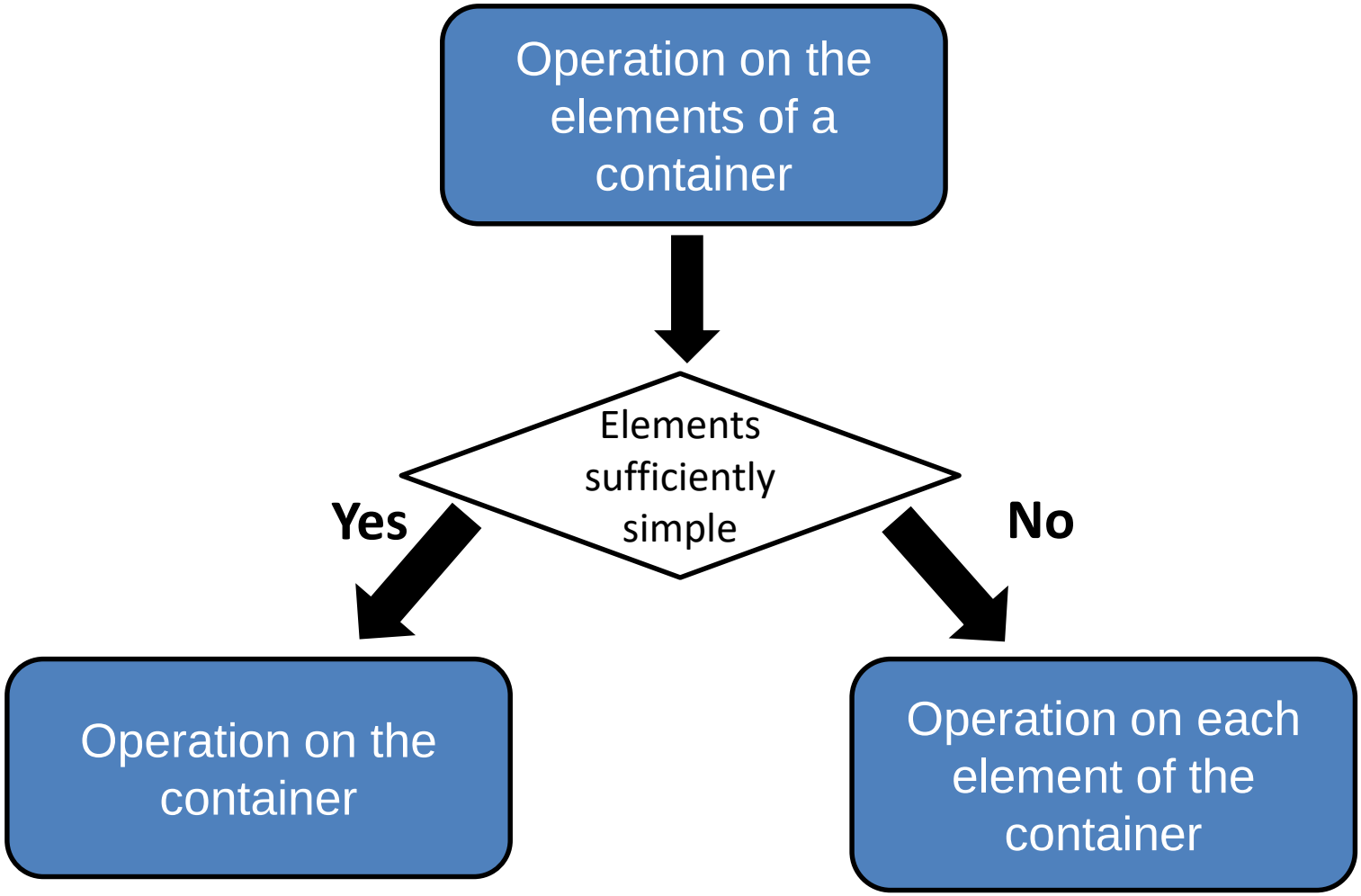


Type-Traits: Optimization

- Code that optimizes itself.
  - ➡ A algorithm is chosen dependent on the type of a variable.

- The STL has optimized versions of `std::copy`, `std::fill`, or `std::equal`.
  - ➡ Algorithm can directly work on the memory.

Type-Traits: Optimization

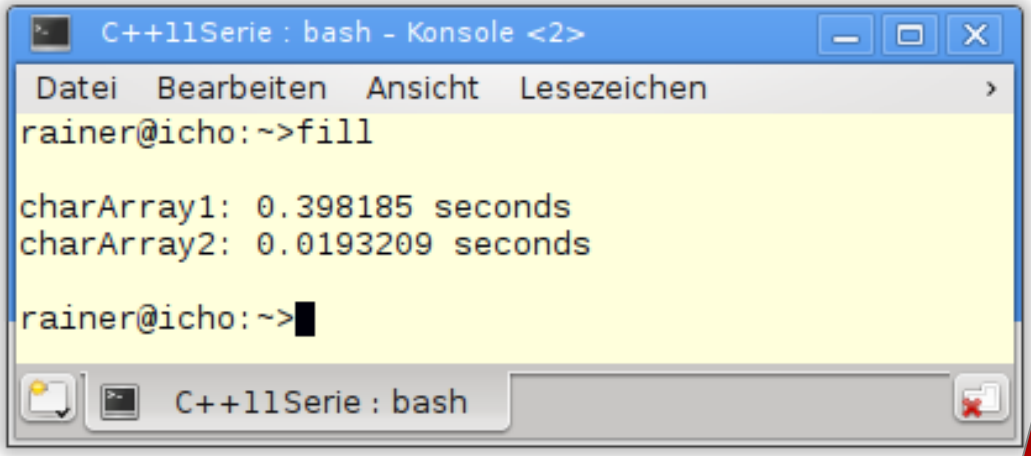


Type-Traits: `std::fill`

```
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`

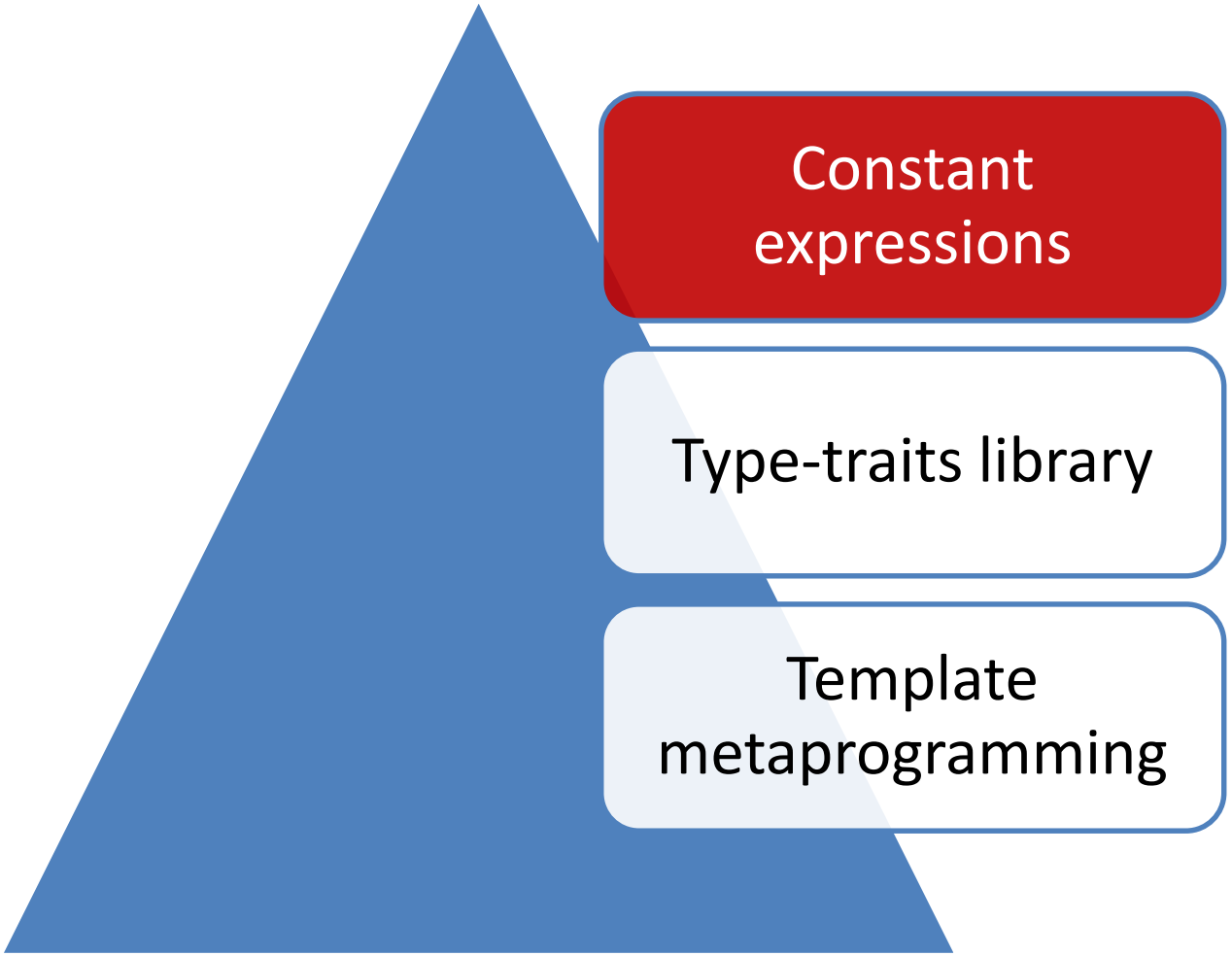
```
const int arraySize = 100'000'000;
char charArray1[arraySize]= {0,};
char charArray2[arraySize]= {0,};

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);
 }
```

Constant Expressions



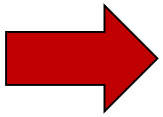
Constant Expressions

- Constant expressions
  - can be evaluated at compile time.
  - give the compiler deep insight into the code.
  - are implicitly thread-safe.
  - are available as variables, user-defined types, and functions.

➡ Support the programming an compile time in an imperative style.

Constant Expressions

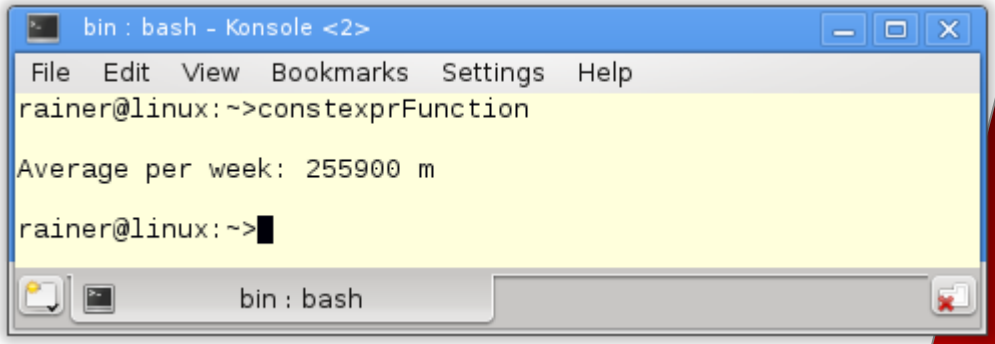
```
#include <iostream>


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



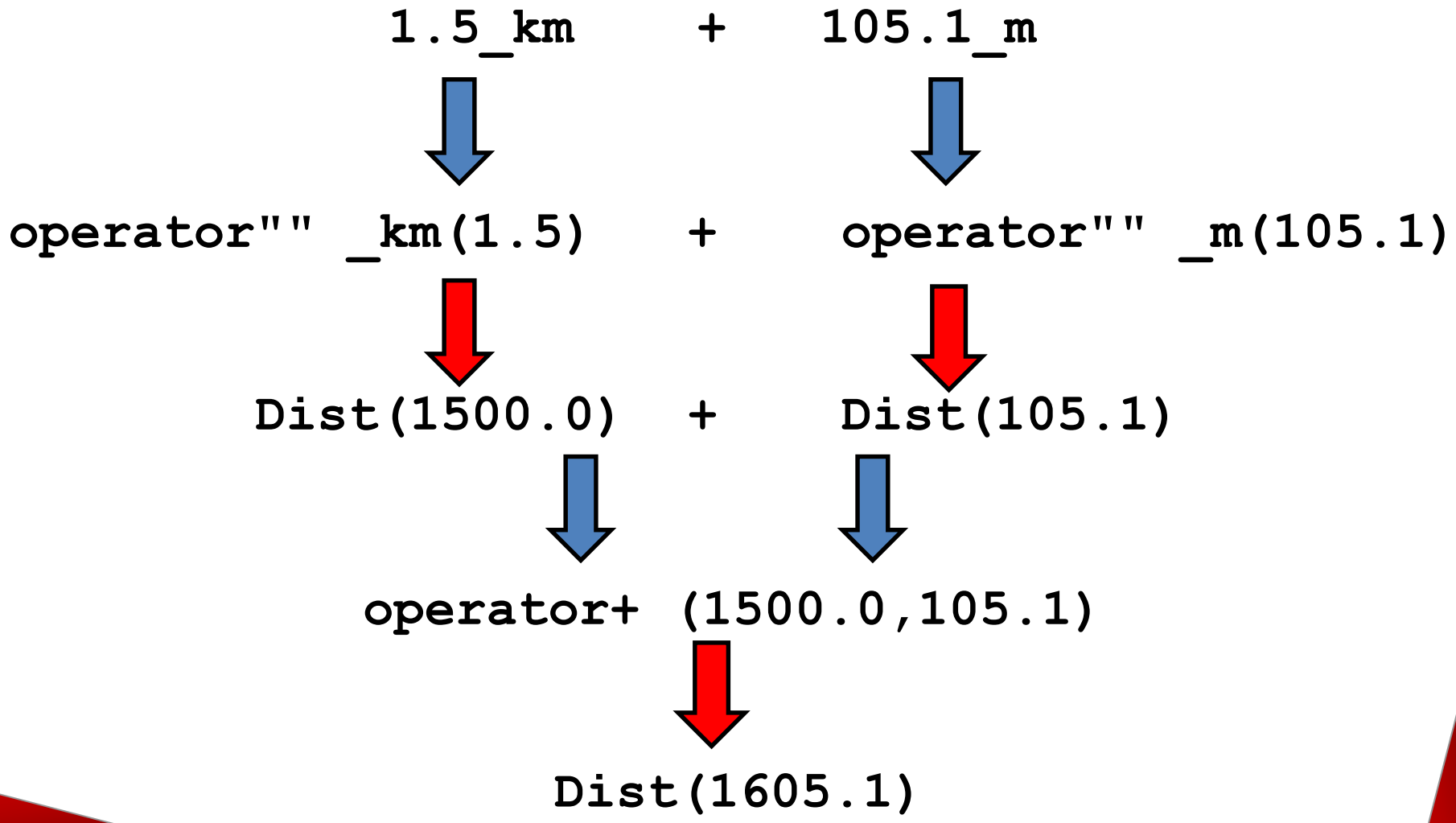
```
mov $0xb,%esi
mov $0x601080,%edi
...
mov $0x64,%esi
mov $0x601080,%edi
...
```

Constant Expressions

```
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 averagePerWeek=  getAverageDistance({distPerWeek1,
                                  distPerWeek2, distPerWeek3, distPerWeek4});

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



Constant Expressions



Constant Expressions

```
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();
}
```

Constant Expressions

```
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;
};
```

Constant Expressions

```
int main(){
  constexpr auto schoolHour= 45min;
  constexpr auto shortBreak= 300s;
  constexpr auto longBreak= 0.25h;
  constexpr auto schoolWay= 15min;
  constexpr auto homework= 2h;
  constexpr auto dayInSec= 2*schoolWay + 6*schoolHour +
                           4*shortBreak + longBreak + homework;

  constexpr chrono::duration<double, ratio<3600>> dayInHours = dayInSec;
  constexpr chrono::duration<double, ratio<60>> dayInMinutes = dayInSec;

  cout << "School day in seconds: " << dayInSec.count();     //  27300
  cout << "School day in hours: " << dayInHours.count();     // 7.58333
  cout << "School day in minutes: " << dayInMinutes.count(); // 455

}
```

Constant Expressions

Characteristics	Template metaprogramming	Constant expressions
Execution time	Compile time	Compile time and run time
Arguments	Types and values	Values
Programming Paradigm	Functional	Imperativ
Modification	No	Yes
Control structure	Recursion	Conditions and Loops
Conditional execution	Template specialization	Conditional statement

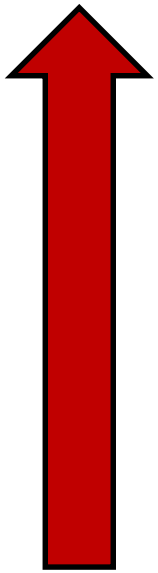
Constant Expressions


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

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

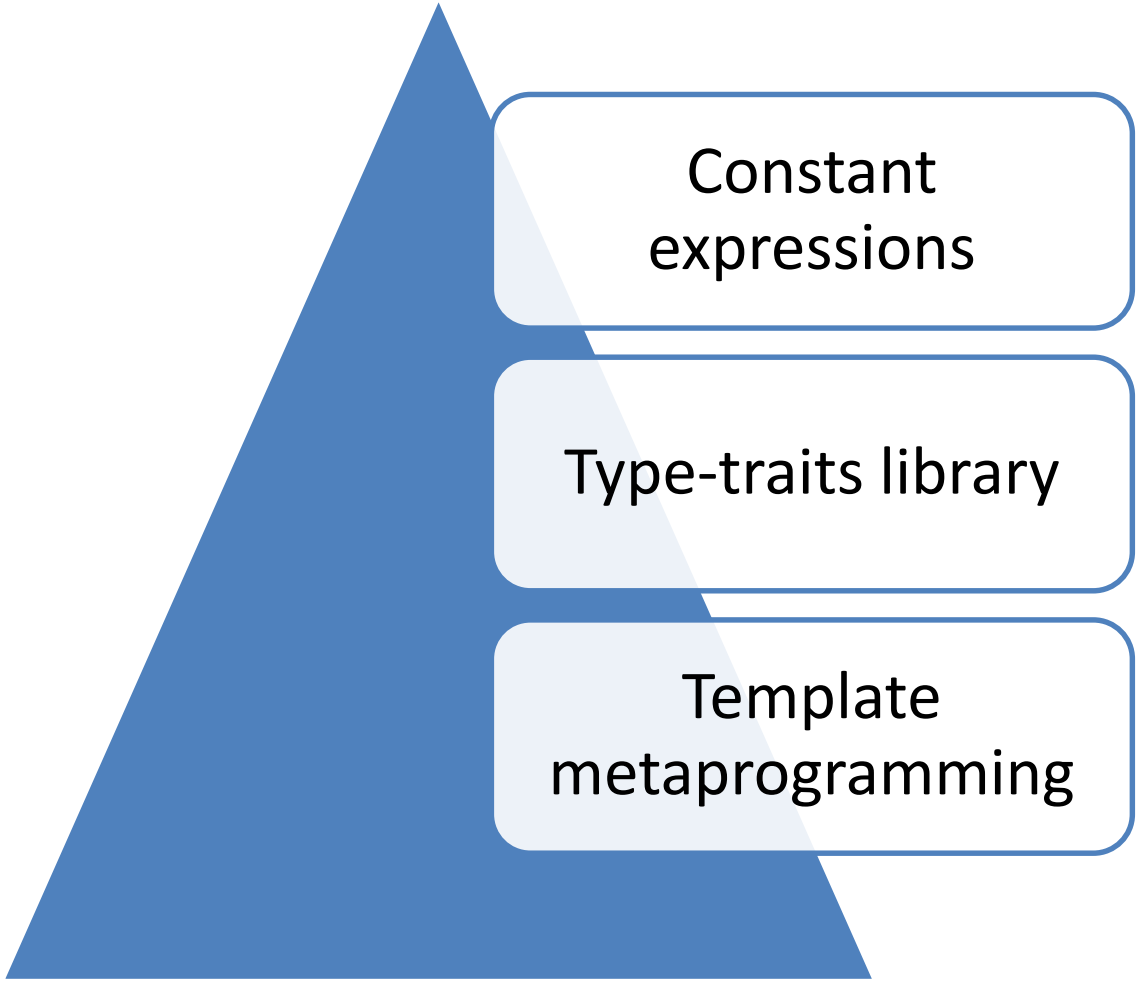
```
pushq    %rbp
movq     %rsp, %rbp
movabsq  $4612811918334230528, %rax
movq     %rax, -8(%rbp)
movl     $0, %eax
popq     %rbp
```

Conclusion

Imperative



Functional



Blogs

www.grimm-jaud.de [Ger]

www.ModernesCpp.de

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.de