Writing C++ metafunctions procedurally

Jyrki Katajainen

[1] *Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark* jyrki@di.ku.dk
http://hjemmesider.diku.dk/~jyrki/
[2] *Jyrki Katajainen and Company, 3390 Hundested, Denmark*

Abstract. In this paper, we describe the design and implementation of the compile-time functions (metafunctions) available in the CPH MPL (Copenhagen metaprogramming library) and the CPH STL (Copenhagen standard template library). Our main point is to show that, after the addition of constant-expression functions to C++, the users need to write very few template metaprograms. This is a good thing when it comes to program readability and maintainability. When explaining the design, we dissect the implementation of four compile-time functions:

Numeric function: `lg`: integer → unsigned integer
Effect: Compute the whole-number binary logarithm of the absolute value $|\mathtt{x}|$ of an integer `x`, i.e. $\lfloor \log_2 |\mathtt{x}| \rfloor$.

Type attribute: `width`: type → unsigned integer
Effect: Return the number of bits in the value representation of an object of the specified type.

Type association: `difference`: type → associated type
Effect: Provide an alias for the difference type to be used for recording the distance between two values specified by the given type.

Type selection: `first_wide_enough`: typelist, integer → type
Effect: Return the first type in the given list of types whose width is larger than or equal to the given integer.

As a small side result we show that, for an integer `x` of type `cphstl::integer<>`, the whole-number binary logarithm of its absolute value $|\mathtt{x}|$, i.e. $\lfloor \log_2 |\mathtt{x}| \rfloor$, can be computed with a constant number of word operations, even if the size of the number is unlimited.

Keywords. Compile-time computations; templates; constant-expression functions

1. Introduction

The C++ programming language [7] is a multi-layer language that allows the programmer to specify actions to be done at different points of time. The same language is used to guide (1) the preprocessor that processes the source code before compilation, (2) the compiler that translates the

program into executable form, and (3) the run-time system that executes the instructions in the executable code. In this paper we are interested in compile-time computations and, to be able to distinguish them from the run-time computations, we borrow two definitions from the book by Abrahams and Gurtovoy [1, Section 2.9]:

Metadata: Any value that can be manipulated by the C++ compile-time machinery can be thought of as metadata. The two most common kinds of metadata are types and integer constants (including those of type **bool**).

Metafunction: A function that operates on metadata and can be invoked at compile time.

Let `X`,...,`Z` be some metadata. Assume that (1) `f` is a metafunction, (2) the passed arguments are `X`,..., and (3) the produced result is `Z`. In this paper, we deal with three kinds of metafunctions:

Pure compile-time function: If the function `f` falls into this category, we annotate its signature in the form `f<X,...>` → `Z`. The angle brackets are used to indicate that the template arguments are the function arguments. If the result is a compile-time constant, technically speaking, `f` is not a function, but a variable template. If the result is a type, `f` is an alias template.

Constant-expression function: For a function of this kind, we write its signature in the form `f(X,...)` → `Z`. If the arguments are not known at compile time, a constant-expression function is handled as any normal run-time function. If the function yields a compile-time constant, the result will be inlined at the place of the function call. If the result is a type, we rely on our type-capsule protocol to be described in Section 4.

Class template as a function: In this case, `f` is written as a class template and the template arguments are the passed arguments. The produced result is obtained via a **static constexpr** variable member—`value`—if the function yields a value and via a nested type member—`type`—if the result is a type. To highlight the difference from the previous two, we annotate the signature of such a metafunction in the form `f<X,...>` :: → `Z`.

The constant-expression functions are the key enablers that make it possible for us to write metafunctions procedurally. The exact rules, what is allowed and what is not allowed in such functions, are given in the C++ standard [2, Clause 9.2.5]. By experience, we have learnt that there are some basic do-and-don't-do rules when writing constant-expression functions:

- The variables should be carefully initialized.
- The functions should have no side-effects. References can only be read, not written.
- It is not allowed to do reinterpret casts.
- Dynamic memory management is not allowed.
- All called functions must also be **constexpr** functions. One cannot use embedded assembler instructions, built-in functions, or C functions.

In the header <type_traits>, the C++ standard library provides a rich set of useful metafunctions. The metafunctions are provided as *type traits* following the class-template-as-a function idiom. Additionally, shortcuts are provided in the way that the suffix _v is added to the name of a metafunction, if it yields a value, and the suffix _t, if the metafunction produces a type.

Metafunctions that operate on types can be classified into several categories based on the purpose, for which they are designed. Some basic (overlapping) categories include the following:

Type predicate: Given a type, a type predicate returns **true** or **false** depending on whether or not a certain condition is met, some operation is supported, or some member (variable, type, function, or class) exists.

Type attribute: Given a type, this kind of metafunction returns a value that represents a specific property of that type.

Type function: This kind of metafunction takes at least one type argument or produces at least one type as a result.

The constant-expression functions suit naturally for the implementation of metafunctions that fail into the first two categories mentioned above. In this paper, we explain how they can also be used for the implementation of other kinds of type functions. The example functions mentioned in the abstract illustrate our design quite exhaustively.

The metafunctions discussed in this paper are part of the CPH MPL (Copenhagen metaprogramming library) and the CPH STL (Copenhagen standard template library). The first release of the CPH MPL is documented in [3] and the second release in [4]. In the first release, most metafunctions were written using the traditional "execution path selection with partial specialization" implementation approach [8, Section 8.3]. In the second release, the metafunctions were refactored to use procedural metaprogramming as explained in this paper. The CPH MPL was developed in parallel with the class templates designed for the manipulation of multiple-precision integers [5]. So the reader must have this limitation in mind.

In the CPH MPL, all traits are given in the **namespace** cphmpl::traits and all constant-expression functions in the **namespace** cphmpl::functions. In addition to this, the shortcuts—variable templates and alias templates—are provided in the **namespace** cphmpl. In all three places. the names of the metafunctions are the same, but the way how they are used differ.

In C++20 [2], the definitions of function templates and class templates can be supplemented by a set of constraints that template arguments have to fulfil before the templates will be instantiated. To specify such constraints, there is a need for a rich set of type predicates, type attributes, and type functions that can be evaluated at compile time. It is quite probable that the standard library cannot meet all needs, so the users must be able to write their own metafunctions. Our hope is that this paper will fill this educational vacuum.

In the rest of this paper, we study procedural metaprograms in greater detail. The presentation is driven by the examples mentioned in the abstract:

- In Section 2, we use the metafunction `lg`, which computes the whole-number binary logarithm of an integer, as an example of a numeric compile-time computation.
- In Section 3, we use the metafunction `width`, which computes the bit width of a type, as an example of a type attribute that yields an integer.
- In Section 4, we use the metafunction `difference`, which determines a type affiliated to a given type, to show how **constexpr** functions can be used to implement metafunctions that produce a type.
- In Section 5 we use the metafunction `first_wide_enough`, which finds the first wide-enough type from a list of types, to show how procedural metaprogramming can be used for a task that has traditionally been solved using template metaprogramming.
- In Section 6, we compare template and procedural metaprogramming. In general, the procedural code is shorter. Our hope is that it would also be easier to maintain. The most important thing is that the code is self-documenting. When the code is readable, it will also be easier to see the vulnerabilities hidden in the implementation.

2. Manipulating integers

Let us begin with a simple numeric compile-time computation that maps an integer to some other integer. We can write such a function (1) as a pure compile-time function or (2) as a constant-expression function. As an example we consider the following numeric computation:

Numeric function: `lg`: integer → **unsigned** integer

Effect: Compute the whole-number binary logarithm of the absolute value |`x`| of an integer `x`, i.e. $\lfloor \log_2 |x| \rfloor$; return 0, if `x` is zero.

In the header file <`cmath`> there is a function `std::ilogb()` which is designed for this purpose. However, as a C function, it is not a **constexpr** function. In its (decorated) interface the keyword **constexpr** is missing:

```
#include <type_traits> // std::is_integral_v

namespace std {

  template<typename T>
  requires std::is_integral_v<T>
  int ilogb(T x);
}
```

Depending on the needs, two possible remedies are offered in the CPH MPL and the CPH STL. In the header file `cphmpl/functions.h++`, a pure compile-time function is provided with the following interface:

```
#include <cstddef> // std::size_t

namespace cphmpl {
```

```
  template<auto x>
  constexpr std::size_t lg = implementation-defined;
}
```

In the header file `cphstl/arithmetics.h++`, a constant-expression function is provided that has the interface:

```
#include "cphmpl/functions.h++" // cphmpl::is_integer
#include <cstddef> // std::size_t

namespace cphstl {

  template<typename T>
  requires cphmpl::is_integer<T>
  constexpr std::size_t lg(T const& x) noexcept;
}
```

So the main difference is how the argument is given.

The pure compile-time function has the drawback that the type of the argument must be one of the built-in integer types since only those are allowed in the template parameters. The superiority of the constant-expression function is illustrated with an example in Figure 1. For the `cphstl::lg()` function, the type of the argument can any of the following:

- standard integer type;
- GNU extension type (**unsigned** `__int128`, **signed** `__int128`);
- class type that wraps a **static** constant (e.g. `std::integral_constant`, `cphstl::constant`);
- integer class type (e.g. `cphstl::ℕ`, `cphstl::ℤ`, `cphstl::integer`).

In Figure 1, on line 25, a user-defined literal is used to create a natural number of type `cphstl::ℕ<156>`. Even if the bit width of this number is 156, only 129 of these bits are in use. The reason, why the **operator**""_10 created a 156-bit number, was that the memory reservation was done conservatively. Any 39-digit decimal number can be represented with 156 bits ($39 \times \lceil \log_2(10) \rceil = 156$). Even if the actual representation turned out to be shorter, this was not fixed later. By declaring the variable `two_to_128` to be of type **auto**, the programmer does not need to worry about this implementation detail.

The implementation of the **constexpr** function `cphstl::lg()` is shown in Figure 2. The key idea is to rely on the connection that, for a positive integer N, the length of its binary representation is $\lfloor \log_2(N) \rfloor + 1$. In the CPH STL we already have a unit-aware function `cphstl::size<>()` which, with a template argument **bool**, returns the number of bits in the binary representation of the given integer. By subtracting one from the result computed by this function, the correct answer for the `lg` computation is obtained, except for $N = 0$.

Let us now analyse the computational complexity of this function. When doing this, we use the parameters:

N**:** The value of the (positive) integer in question.

```
#include "cphstl/arithmetics.h++" // cphstl::lg
#include "cphstl/constants.h++" // cphstl::constant
#include <type_traits> // std::integral_constant

int main() {
  // use for a constant

  using four = cphstl::constant<4>;
  constexpr auto two = cphstl::lg(four());
  static_assert(two == 2);

  using ten = std::integral_constant<int, 10>;
  constexpr auto three = cphstl::lg(ten());
  static_assert(three == 3);

  // use for an int

  static_assert(cphstl::lg(511) == 8);
  static_assert(cphstl::lg(512) == 9);

  // use for a CPH STL integer cphstl::ℕ<156>

  using namespace cphstl::literals;

  constexpr auto two_to_128 = "340282366920938463463374607431768211456"_10;
  static_assert(cphstl::lg(two_to_128) == 128);

  return 0;
}
```

Figure 1. Some examples of the use of the **constexpr** function `cphstl::lg()`. **Source:** file `Playground/lg.c++`

n**:** The number of bits in the binary representation of N.
ℓ**:** The number of words used for storing N.
w**:** The number bits in each such word.

As a natural unit to express the complexity, we use the number of word operations performed. Often, in the algorithmic literature, this is referred as the running time of an algorithm.

For a **static** constant, the running time is determined by the loop executed in the function `cphstl::functions::lg()`. Clearly, this requires $O(n)$ word operations. Since $n \leq w$, this running time—when done at compile time—is acceptable. The compiler will place the result at the point, where the function was called, so the resulting run-time cost is $O(1)$ word operations. For non-**static** data, the running time depends on the implementation details of the **constexpr** functions `cphstl::abs()` and `cphstl::size<>()`, and the type of the underlying integer. Next, we consider the two most interesting

```
#include "cphmpl/functions.h++" // cphmpl::make_unsigned cphmpl::is_constant ...
#include <cstddef> // std::size_t

namespace cphstl::functions {

  template<auto x>
  constexpr auto lg() {
    using Z = decltype(x);
    using N = cphmpl::make_unsigned<Z>;
    N n = (x < 0) ? N(- x) : N(x);
    N result = 0;
    N i = n;
    while (i > 1) {
      i = i / 2;
      ++result;
    }
    return result;
  }
}

namespace cphstl {

  template<typename T>
  requires cphmpl::is_constant<T>
  constexpr auto lg(T const& x) noexcept {
    constexpr auto result = functions::lg<x()>();
    return result;
  }

  template<typename T>
  requires cphmpl::is_integer<T> and not cphmpl::is_constant<T>
  constexpr std::size_t lg(T const& x) noexcept {
    using U = cphmpl::make_unsigned<T>;
    U y = cphstl::abs(x);
    std::size_t bits_in_use = cphstl:: template size<bool>(y);
    if (bits_in_use == 0) {
      return std::size_t(0);
    }
    return bits_in_use - 1;
  }
}
```

Figure 2. Two overloaded versions of the constexpr function cphstl::lg(). For integers x and y, the constexpr function cphstl::abs(x) computes the absolute value of x and the constexpr function cphstl::size<bool>(y) computes the size of y measured in bits. **Source:** file cphstl/arithmetics.h++

cases: the class templates `cphstl::integer` and `cphstl::ℤ`.

Let `W` denote the type of the words (sometimes called the limbs) used in the implementation of an integer of type `cphstl::integer<W>`. The actual representation is an array of such words. Internally, the integers are represented using the sign-magnitude representation, so the sign of the number and the length of the array are maintained separately. Moreover, the arithmetic operations are required to keep the integers in sanitized form so that there are only some padding bits in one word at the most significant end of the representation. Under these circumstances, (1) the absolute value computation in Figure 2 on line 31 requires a sign flip which costs $O(1)$ word operations, and (2) the size computation on line 32 requires the determination of the number of leading zero bits (`nlz`) in the most significant word of the representation. Since the integer knows its length ℓ and the bit width w of `W` is readily available (see Section 3), the number of bits in use is $\ell \times w - \texttt{nlz}$. The number of leading zero (one) bits in a word can be computed using a special instruction, which is found on most computers. In the CPH STL in the file `cphstl/bit_tricks.h++`, there are special routines that can utilize the underlying hardware capabilities. Observe, however, that these routines are not **constexpr** functions. When the special instruction is available, the cost of the function `cphstl::lg()` is $O(1)$ word operations. Without this instruction, the cost would be $O(w)$ word operations, which is still acceptable, since the integer can be of unlimited size. The important point is that the running time is independent of the length of the input number.

An integer of type `cphstl::ℤ<b>` is also represented as an array of words but, since the bit width `b` is specified in the code, the length of this array will be fixed at compile time. Internally, these integers are represented in two's complement form. The sign is determined by inspecting the most significant word; this involves $O(1)$ word operations. If the number is negative, in the `abs` computation all bits are complemented and one is added to the complemented result. The cost of this is $O(\ell)$ word operations. For a positive number, in the `size` computation the number of leading zeros is determined. For a negative integer, the `size` computation is even more expensive, since the number of leading ones is computed and then, to get the exact length of the representation, it is checked whether the remaining bits are all zeros. If this is the case, after the complementation, the addition will overflow and the representation will become one longer. To sum up, for this integer type, the worst-case running time of the function `cphstl::lg()`—measured as the number of word operations—is linear on the length of the input number.

3. Querying properties

In this section, we examine one specific metafunction `cphmpl::width<>` which is one of the most used metafunctions of the CPH MPL. Probably, it should even deserve the same status as the built-in **operator sizeof()**. The specification of this metafunction is as follows:

Type attribute: `width`: type → **unsigned** integer

Effect: Return the number of bits in the value representation of an object of the given type; this number is indeterminate for an unbounded type. The object representation can be larger. For example, there can be some padding bits due to alignment.

In the header file `cphmpl/functions.h++`, the actual computation is done by a **constexpr** function and two additional interfaces are provided: a pure compile-time function and a traits template that uses the `type`/`value` naming scheme when announcing its results. The three interfaces look like this:

```
#include <cstddef> // std::size_t

namespace cphmpl::functions {

  template<typename T>
  constexpr auto width() noexcept;
}

namespace cphmpl::traits {

  template<typename T>
  class width {
  public:

    using type = implementation-defined;
    static constexpr type value = implementation-defined;
  };
}

namespace cphmpl {

  template<typename T>
  constexpr std::size_t width = implementation-defined;
}
```

For the traits template the produced type is the smallest **unsigned** built-in integer type that can store the result (cf. Section 5).

The implementation of the **constexpr** function `cphstl::functions::width()` is given in Figure 5, and the traits classes and the helper function used by it are shown in Figures 3 and 4, respectively. One problem with this implementation is the switch statement: It is difficult—if impossible—to cover all the relevant cases. For example, the cases for `std::array`, `std::bitset`, and a multi-dimensional C array were added to the current implementation, after we observed that the early implementation in the CPH MPL (release 1) [3] did not produce the correct answer in these cases. Even for a compiler, it might be difficult to determine which part of the object representation belongs to the value representation. Therefore, to be sure that the produced answer is correct, a bounded class type should provide the **static** variable `width`, which is recognized by this (and the early) implementation.

```
#include <array> // std::array
#include <bitset> // std::bitset
#include <cstddef> // std::size_t

namespace cphmpl::traits {

  template<typename T>
  class is_std_array {
  public:
    static constexpr bool value = false;
  };

  template<typename T, std::size_t N>
  class is_std_array<std::array<T, N>> {
  public:
    static constexpr bool value = true;
    static constexpr std::size_t size = N;
    using value_type = T;
  };

  template<typename T>
  class is_std_bitset {
  public:
    static constexpr bool value = false;
  };

  template<std::size_t N>
  class is_std_bitset<std::bitset<N>> {
  public:
    static constexpr bool value = true;
    static constexpr std::size_t width = N;
  };

  template<typename T>
  class has_static_variable_width {
  public:
    static constexpr bool value = false;
  };

  template<typename T>
  requires requires {T::width;}
  class has_static_variable_width<T> {
  public:
    static constexpr bool value = true;
  };
}
```

Figure 3. The traits templates used by the constant-expression function `cphmpl::functions::width()`. **Source:** file `cphmpl/functions.h++`

```
#include <cstddef> // std::size_t
#include <type_traits> // std::is_array_v std::rank_v std::extent_v

namespace cphmpl::functions {

  template<typename T, std::size_t i = 0, std::size_t product = 1>
  requires std::is_array_v<T>
  constexpr std::size_t total_size() {
    if constexpr (i == std::rank_v<T>) {
      return product;
    }
    else {
      constexpr std::size_t up_to_i = product * std::extent_v<T, i>;
      return total_size<T, i + 1, up_to_i>();
    }
  }
}
```

Figure 4. The helper `cphmpl::functions::total_size()` used by the **constexpr** function `cphmpl::functions::width()`. **Source:** file `cphmpl/functions.h++`

Since we rely on the fact that the bounded types know their width, the cost of the `width` computation is $O(1)$ word operations. The only exception is a multi-dimensional array, for which the running time depends on the number of dimensions that the array has.

4. Constructing types

In this section, we consider the task of writing a metafunction that, for a given type, produces some affiliated type. The basic example is the task of producing the difference type associated with an iterator. In the book by Stepanov and Rose [6, Section 10.5], the following code was used to perform this task. Observe that we have modified the code so that it compiles in our computing environment.

```
#include "cphmpl/concepts.h++" // cpp20 and cpp17 concepts
#include <iterator> // std::iterator_traits

namespace cpp17 {

  template<typename I>
  requires cpp17::is_input_iterator<I>
  using difference = typename std::iterator_traits<I>::difference_type;
}
```

We want to generalize this computation to work for every type for which a difference type might be relevant. The abstract specification of this task is as follows:

```
#include <climits> // CHAR_BIT
#include <cstddef> // std::size_t
##include <limits> // std::numeric_limits
#include <type_traits> // std::is_const_v std::is_volatile_v ...

namespace cphmpl {
  constexpr std::size_t indeterminate = std::numeric_limits<std::size_t>::max();
}

namespace cphmpl::functions {

  template<typename T>
  constexpr auto width() noexcept {
    if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
      return width<std::remove_cv_t<T>>();
    }
    else if constexpr (std::is_reference_v<T>) {
      constexpr std::size_t result = CHAR_BIT * sizeof(T);
      return result;
    }
    else if constexpr (std::is_pointer_v<T>) {
      constexpr std::size_t result = CHAR_BIT * sizeof(T);
      return result;
    }
    else if constexpr (std::is_arithmetic_v<T>) {
      constexpr std::size_t sign = std::numeric_limits<T>::is_signed;
      constexpr std::size_t digits = std::numeric_limits<T>::digits;
      constexpr std::size_t result = sign + digits;
      return result;
    }
    else if constexpr (cphmpl::is_gnu_extension<T>) {
      constexpr std::size_t result = CHAR_BIT * sizeof(T);
      return result;
    }
    else if constexpr (traits::has_static_variable_width<T>::value) {
      return T::width;
    }
    else if constexpr (traits::is_std_bitset<T>::value) {
      constexpr std::size_t result = is_std_bitset<T>::width;
      return result;
    }
    else if constexpr (traits::is_std_array<T>::value) {
      using V = typename is_std_array<T>::value_type;
      constexpr std::size_t result = is_std_array<T>::size * width<V>();
      return result;
    }
    else if constexpr (std::is_array_v<T>) {
      using V = std::remove_all_extents_t<T>;
      constexpr std::size_t result = total_size<T>() * width<V>();
      return result;
    }
    else { // default
      return indeterminate;
    }
  }
}
```

Figure 5. Implementation of the constexpr function cphmpl::functions::width().
Source: file cphmpl/functions.h++

Type association: `difference`: type → associated type
Effect: Provide an alias for the difference type to be used for recording the distance between two values specified by the given type. For a class type `T`, the result is `T::difference_type` if this type member exists.

We use this task to illustrate the protocol how we write type functions in the CPH MPL. The simple, but crucial, observation is that any function returning a value also returns the type of this value. This type can be yielded using the introspection facility **decltype**.

The key tool in our construction is the concept of a *type capsule* (see Figure 6). This is a class template that has a trivial constructor and a single type member `type`. This nested type recalls the template argument given when an instance of the type capsule is created. Now the idea is that the callee puts the return type inside a capsule, returns an instance of that capsule to the caller, and the caller opens the capsule using **decltype** and then retrieves the type inside the capsule. In this way, any type function can be implemented as a **constexpr** function.

```
namespace cphmpl::functions {

  template<typename T>
  class capsule {
  public:

    capsule() {
    }

    using type = T;
  };
}
```

Figure 6. Implementation of the type-capsule class template. **Source:** file `cphmpl/functions.h++`

Assume that we use some pointer type to refer to the values stored in the address range $[0\,..\,N)$. The main reason, why the `difference` computation is needed, is that the range of the signed integers recording the distance specified by two pointers is $[-N\,..\,N]$. So the binary representation of the difference type is wider than that of the size type. (Since we allow a pointer to the past-the-end element, the representation can be two bits longer.) As Stepanov and Rose [6, footnote on p. 187] remarked, some earlier versions of the Intel architectures supported short pointers and long pointers so it was important to use the correct difference type. Today, for most uses the type `std::ptrdiff_t` is sufficient.

```
#include <type_traits> // std::is_object_v

namespace cphmpl::traits {

  template<typename T>
  class is_object_pointer {
  public:
    static constexpr bool value = false;
  };

  template<typename T>
  requires std::is_object_v<T>
  class is_object_pointer<T*> {
  public:
    static constexpr bool value = true;
  };

  template<typename T>
  class has_type_member_difference_type {
  public:
    static constexpr bool value = false;
  };

  template<typename T>
  requires requires {typename T::difference_type;}
  class has_type_member_difference_type<T> {
  public:
    static constexpr bool value = true;
  };
}
```

Figure 7. Traits templates used in the implementation of the constexpr function cphmpl::functions::difference( ). **Source:** file cphmpl/functions.h++

```
#include "cphstl/integers.h++" // cphstl::ℤ cphstl::integer
#include <cstddef> // std::ptrdiff_t
#include <iterator> // std::iterator_traits
#include <type_traits> // std::is_const_v std::is_volatile_v ...
#include <utility> // std::declval

namespace cphmpl::functions {

  template<typename T>
  constexpr auto difference() noexcept {
    if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
      return difference<std::remove_cv_t<T>>();
    }
    else if constexpr (std::is_reference_v<T>) {
      return capsule<std::ptrdiff_t>();
    }
    else if constexpr (traits::is_object_pointer<T>::value) {
      return capsule<std::ptrdiff_t>();
    }
    else if constexpr (cphmpl::is_iterator<T>) {
      using R = typename std::iterator_traits<T>::difference_type;
      return capsule<R>();
    }
    else if constexpr (traits::has_type_member_difference_type<T>::value) {
      using R = typename T::difference_type;
      return capsule<R>();
    }
    else if constexpr (requires(T const& a, T const& b) {std::is_integral_v<decltype((a
    ↪ - b))>;}) {
      using D = decltype(std::declval<T>() - std::declval<T>());
      constexpr std::size_t w = cphmpl::width<D>;
      using R = cphstl::ℤ<w + 2>;
      return capsule<R>();
    }
    else { // default
      using R = cphstl::integer<unsigned long long int>;
      return capsule<R>();
    }
  }
}
```

Figure 8. Implementation of the function `cphmpl::functions::difference()`.
Source: file `cphmpl/functions.h++`

A trivial solution would be to define the difference type as an alias of `cphstl::integer<>` which is an integer type with unlimited precision. However, in the CPH MPL, we respect the decisions made by other developers so we use a difference type if such is specified as a nested member. The implementation of the **constexpr** function `cphmpl::functions::difference()` is shown in Figure 8. The traits classes in Figure 7 were needed to make the **if constexpr** statements SFINAE-friendly (for more details, see e.g. [8, Section 8.4]).

As shown in Figure 8, the function `cphmpl::functions::difference()` does its job by a compile-time switch statement. There are the following cases:

(1) For a **const** or **volatile** type, the function is called recursively after removing the qualifier(s) (lines 11–12).
(2) For a reference type, the output is `std::ptrdiff_t` since, in principle, a reference is a **const** pointer (lines 14–15).
(3) For an object pointer, the output is `std::ptrdiff_t` (lines 17–18).
(4) For an iterator, the output is retrieved using the `std::iterator_traits` facility (lines 20–22).
(5) For a type that has the nested type member `difference_type`, the output is this type as such (lines 24–26).
(6) For an integer-like type that supports **operator**−, the output is a signed integer that is two bits wider (lines 28–32).
(7) For any other type, the output is the type `cphstl::integer<unsigned long long int>` which provides integers of unlimited precision (lines 34–36).

Using this **constexpr** function, the type function can be defined to be the following alias template:

```
namespace cphmpl {

  template<typename T>
  using difference = typename decltype(functions::difference<T>())::type;
}
```

5. Selecting types

In this section, we consider a task where we are given a list of types and an integer, and we want to select one of these types according to a criterion based on the integer. As a concrete example, let us extend the `cphmpl::traits::lg` traits so that the result is given in the smallest integer type that can store it. For a **static** constant `x`, we could compute its bit width $b \stackrel{\text{def}}{=} \lfloor \log_2 |x| \rfloor + 1$ at compile time and then store the computed result in `cphstl::ℕ<b>`. However, some users may want to get the result in the best-fitting built-in integer type instead, so we end up with the following specification.

Type selection: `first_wide_enough`: typelist, integer → type

Effect: Return the first type in the given list of types whose width is larger than or equal to the given integer.

In the CPH MPL, there is a pure compile-time function that does this computation; it has the following interface:

```
#include "cphmpl/lists.h++" // cphmpl::is_typelist

namespace cphmpl {

  template<typename L, std::size_t width>
  requires cphmpl::is_typelist<L>
  using first_wide_enough =
    typename decltype(functions::first_wide_enough<L, width>())::type;
}
```

Here we have also revealed the implementation. In principle, we would need a `for` loop that traverses through the list `L` of types and stops when the first type that is wide enough is found. The point is that a `for` loop cannot be used since the loop variable is not a constant expression. In the implementation described in Figure 9, the `for` loop is implemented using recursion. By this program the list is traversed recursively and, if the head of the list is not wide enough, the recursion proceeds to the tail of the list.

Now it is in place to analyse the running time of this type-selection computation—measured as the number of word operations executed by the compiler. Of course, the run-time cost is non-existent since the work is done at compile time. Let ℓ denote the length of the typelist `L`. In the recursive loop, each type is visited exactly once. In addition to the original typelist, in Figure 9, on line 13 we instantiate the present type of the typelist and on line 14 a typelist that is one shorter than the original. In the worst case, $O(\ell)$ class-template instantiations are done. There are also some hidden instantiations inside the nullary metafunctions `L::front` and `L::pop_front`, but asymptotically the number of instantiations remains the same.

It is well-known that template instantiations are costly. As Vandevoorde et al. [8, Section 23.3] pointed out in their book on C++ templates, even for relatively modest class templates, a compiler may allocate over a kilobyte of storage. If the storage between different instantiations is not shared, the amount of allocated storage will depend on the number of template arguments. And if these are copied, the amount of storage allocated by the function doing the type-selection computation is $\Theta(\ell^2)$ words. This means that the running time must be quadratic, too. Thus, to speed up the compilation process, the naive copying of the template arguments should be avoided by all possible means.

6. Comparing the implementation approaches

Some of the type functions provided in the CPH MPL are also specified in the draft of the C++20 standard [2, Clause 23.3.2] in the form of traits templates. This gives us a good opportunity to compare template metaprogramming and procedural metaprogramming when it comes to defining a type function. In the standard, the counterpart of the metafunction `cphmpl::difference<>`

```
#include <cstddef> // std::size_t
#include "cphmpl/lists.h++" // cphmpl::is_typelist cphmpl::size

namespace cphmpl::functions {

  template<typename L, std::size_t bit_width>
  requires cphmpl::is_typelist<L>
  constexpr auto first_wide_enough() {
    if constexpr (cphmpl::size<L> == 0) {
      return capsule<void>();
    }
    else {
      using head = typename L::front;
      using tail = typename L::pop_front;
      constexpr std::size_t present = cphmpl::width<head>;
      if constexpr (present ≥ bit_width) {
        return capsule<head>();
      }
      else {
        return first_wide_enough<tail, bit_width>();
      }
    }
  }
}
```

Figure 9. Finding the first type in list `L` whose width is larger than or equal to the given integer. The class template `cphmpl::typelist` is implemented in the file `cphmpl/typelist.h++`. This data structure supports normal list operations, like `L::front` and `L::pop_front`, that are executed at compile time. The compile-time predicate `cphmpl::is_typelist<>` and function `cphmpl::size<>` are implemented in the file `cphmpl/lists.h++`. **Source:** file `cphmpl/functions.h++`

is the metafunction `cpp20::incrementable_traits<>::`. To keep this paper self-contained, we have reproduced the implementation of this class template in Figure 10; again some minor modifications have been done to the original in order to be able to compile the code in our environment.

For iterators, the usage of the these two facilities is about the same:

```
using I = /* some iterator */;
using X = cpp20::iter_difference_t<I>;
using Y = cphmpl::difference<I>;
```

The traits template also works for other data types, like containers, but the shortcut `cpp20::iter_difference_t<>` is only defined for iterators.

In the traditional approach, a traits-based metafunction is constructed following the principles set out below:

(1) The primary template is defined that gives the default behaviour. For `cpp20::incrementable_traits<>::` (Figure 10, line 8), the default behaviour is that the member `difference_type` is undefined.

```
#include <cstddef> // std::size_t std::ptrdiff_t
#include <type_traits> // std::is_object_v std::is_integral_v ...
#include <utility> // std::declval

namespace cpp20 {

  template<typename>
  struct incrementable_traits {
  };

  template<typename T>
  requires std::is_object_v<T>
  struct incrementable_traits<T*> {
    using difference_type = std::ptrdiff_t;
  };

  template<typename I>
  struct incrementable_traits<I const>
    : incrementable_traits<I> {
  };

  template<typename T>
  requires requires { typename T::difference_type; }
  struct incrementable_traits<T> {
    using difference_type = typename T::difference_type;
  };

  template<typename T>
  requires (not requires {typename T::difference_type;} and
    requires(T const& a, T const& b) {std::is_integral_v<decltype((a - b))>;})
  struct incrementable_traits<T> {
    using difference_type = std::make_signed_t<decltype(std::declval<T>() -
    ↪ std::declval<T>())>;
  };

  template<typename T>
  using iter_difference_t = typename
    ↪ incrementable_traits<std::remove_cvref_t<T>>::difference_type;
}
```

Figure 10. Implementation of the metafunction `incrementable_traits<>::` taken from the draft of the C++20 standard [2].

(2) Specializations of the primary template are defined to specify the behaviour for some specific types. For `cpp20::incrementable_traits<>::`, special treatment is given for object pointers (Figure 10, line 13), for types that have the nested type member `difference_type` (Figure 10, line 24), and for integer-like types that support **operator**− (Figure 10, line 31).

Input	Output cphmpl::difference<>	Output cpp20::incrementable_traits<>:: ↪ difference_type
default	cphstl::integer<>	undefined
int&	std::ptrdiff_t	int
unsigned int	cphstl::ℤ<34>	int

Figure 11. Behaviour of the metafunction cphmpl::difference<> and the traits-based metafunction cpp20::incrementable_traits<>:: for some specific inputs.

(3) For some types, metafunction forwarding [8, Section 19.3.2] can be used to inherit the type member difference_type from one of the other type traits rather than declaring its own type member. For cpp20::incrementable_traits<>::, this technique is used for const types (Figure 10, line 18).

It should be pointed out that the metafunction cphmpl::difference<> and the traits-based metafunction cpp20::incrementable_traits<>:: do not always produce the same result. This is elaborated in Figure 11. We will leave it for the reader to decide which behaviour is correct.

Due to partial specialization, in the traditional approach the cases must be mutually exclusive. For cpp20::incrementable_traits<>::, this is seen as a condition in one case (Figure 10, line 23) and its negation in the next (Figure 10, line 29). The procedural approach is more relaxed since the cases are processed in the order specified by the program. Simply, the first branch that gives a match is taken. Instead of metafunction forwarding, one could also define the output recursively as in the procedural approach.

In the CPH MPL, the variable templates, alias templates, and constexpr functions are offered to reduce verbosity. By experience, we have learnt that they also have some downsides:

(1) The variable and alias templates cannot be forwarded. We encountered this problem when working with the files cphmpl/functions.h++ and cphstl/integers.h++; the class templates for integers use metafunctions and, as we have seen in this paper, the metafunctions use integers. What is the correct order of defining different utilities? Because the shortcuts cannot be forwarded, we always try to include them first. For a big code base, this ordering issue can become a headache.

(2) The variable and alias templates cannot be specialized, whereas the traits templates can be specialized by users. Our constexpr functions recognize hooks, which make them extensible. The question is whether these hooks are enough. Of course, one can always write a new metafunction that calls the old one to handle the known cases, and extends the behaviour for the cases that are not handled correctly. This is the way most utilities in the CPH MPL have been programmed. The standard-library utilities handle the built-in types smoothly, but in some cases an extension for class types has been necessary.

(3) Some of the metafunctions can be used in three different ways: as a traits class, as a **constexpr** function, or as a shortcut. Mixed use of these alternatives can be confusing.

7. Afterword

In the book on generic programming, Stepanov and Rose [6, Section 10.3] had a short discussion on type functions where they wrote:

> "Unfortunately, mainstream programming languages do not contain type functions, even thought they would be easy to implement."

Now, only a few years later, we can conclude that C++ contains a powerful toolbox for writing type functions. And, as the examples in this paper show, the implementation of the type functions is often straightforward.

Metaprogramming—as programming in general—is an art. Naturally, one may ask how this art can be made more beautiful. Some details could always be improved:

(1) Metadata—types and integer constants—would deserve more uniform handling. For example, it should be possible for a function to return a type so that we do not need to go via the type capsule.

(2) A typelist is a usable compile-time data structure, but an array of types is what is needed.

(3) A compile-time **for** loop as an alternative to recursion would make things even simpler. When such a loop is unrolled, at each iteration step, a separate index variable can be used, it can be initialized using the index variable from the previous step, and it can be kept constant thereafter. So there is no conflict with the **constexpr** thinking.

(4) Let `X` be a built-in integer type, and `Y` and `Z` some arbitrary types. In the CPH STL there are several **constexpr** functions `f` for which two interfaces are provided: `f<X>(Y)` → `Z` and `f(X, Y)` → `Z`. The first form emphasizes that the first argument is always known at compile time, but the implementations of the two are almost identical. This code duplication feels like a waste.

(5) The nature of type functions is slightly different from that of normal functions. For some inputs even undefined behaviour is acceptable, since any incorrect use will be detected at compile time. Sometimes, for a type predicate, we had an urge to use three-value logic since in some cases the result was unknown. For the type function `width`, plus infinity ∞ might have been a better answer than `cphmpl::max<std::size_t>` for an unbounded type. Though, the special values (like `unknown`, `indeterminate`, and **void**) can make the use of the type functions a little more complicated.

Software availability

Much of the code described in this paper has been taken from the CPH MPL (release 2). A reference for the users of this metaprogramming package is provided in [4]—this reference contains the source code in its entirety.

To get a wider picture, look at the documentation for the components of the CPH STL, the CPH LEDA, and the CPH MPL. Normally, the source code of each component is released together with a technical report that gives more details about the underlying implementation. Therefore, to get more information, we refer to the papers, reports, and downloadable `tar` and `zip` archives that can be accessed via the website http://www.cphstl.dk.

References

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Pearson Education, Inc. (2005). https://isbnsearch.org/isbn/978-0-321-22725-6

[2] The C++ Standards Committee, Standard for Programming Language C++, Working draft **N4861**, ISO/IEC (2020).

[3] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Department of Computer Science, University of Copenhagen (2017). http://hjemmesider.diku.dk/~jyrki/Myris/Kat2017R.html

[4] J. Katajainen, Documenting the CPH MPL—release 2—by code, CPH STL report **2020-1**, Department of Computer Science, University of Copenhagen (2020). http://hjemmesider.diku.dk/~jyrki/Myris/Kat2020aS.html

[5] J. Katajainen, A C++ multiple-precision integer package, CPH STL report **2020-3**, Department of Computer Science, University of Copenhagen (work in progress).

[6] A. A. Stepanov and D. E. Rose, *From Mathematics to Generic Programming*, Pearson Education, Inc. (2015). https://isbnsearch.org/isbn/978-0-321-94204-3

[7] B. Stroustrup, *The C++ Programming Language*, 4th edition, Pearson Education, Inc. (2013). https://isbnsearch.org/isbn/978-0-321-56384-2

[8] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*, 2nd edition, Addison-Wesley (2018). https://isbnsearch.org/isbn/978-0-321-71412-1